Analysis and Implementation of Analytical Placement Algorithms for Microelectronic Circuits

Ntentos Stavros

A Thesis presented for the degree of
Electrical and Computer Engineering

UNIVERSITY OF
THESSALY

Supervisors:

Dr. Stamoulis Georgios, Professor
Dr. Nestor Eumorfopoulos, Assistant Professor

Hardware Lab (by **Dr. Sotiriou Christos**)
Volos, Greece

May 2017

Dedicated to

Everyone who tries too much to succeed. Don't worry, sometime it will shine bright.

Everyone who helped me, stood by me and believed in me.

Thank you so much *(next time, please, believe a little less!)*

Me. I failed a lot more than I succeeded.

And this was okay

Ανάλυση και Υλοποίηση Αλγορίθμων Τοποθέτησης Αναλυτικής Μεθοδολογίας για Μικροηλεκτρικά Κυκλώματα

Περίληψη

Στη σημερινή τεχνολογία του **Electronic Design Automation (EDA)**, είναι κρίσιμης σημασίας η κατασκευή νέων και εξελιγμένων **Ολοκληρωμένων Κυκλωμάτων (ICs)** αποδοτικά, αυτοματοποιημένα, και κάποιες φορές με το μικρότερο δυνατό μέγεθος. Υπάρχει πληθώρα αλγορίθμων που χρησιμοποιούνται για να το επιτύχουν αυτό - αλλά όλοι εξυπηρετούν τον ίδιο κοινό σκοπό: Να δώσουν ζωή σε μία φυσική οντότητα μέσω κώδικα, και να το υλοποιήσουν. Είναι σύνηθες να διαιρούμε υψηλού επιπέδου διαδικασίες σε μικρότερες, και το **EDA** δεν αποτελεί εξαίρεση.

Στο πρώτο βήμα της υλοποίησης του σχηματικού (γνωστό και ως **Τοποθέτηση**), ο στόχος είναι να τοποθετηθούν τα στοιχεία με τέτοιο τρόπο έτσι ώστε να ελαχιστοποιηθούν διάφορες μετρικές-στόχοι. Απλοί στόχοι, όπως παραδείγματος χάριν ελάχιστο μήκος καλωδίων, αλλά και σύνθετες, όπως το να απαγορεύεται η επικάλυψη των στοιχείων αναμεταξύ τους. Μετά την ολοκλήρωση αυτού του σταδίου, ο σχεδιασμός του IC προετοιμάζεται για τα επόμενα στάδια της διαδικασίας του **EDA** που είναι το **Physical Verification and Signoff**, η **Κατασκευή**, και τέλος **Συσκευασία και Έλεγχος**

Μέχρι στιγμής, **Στοχαστικές** μέθοδοι έχουν χρησιμοποιηθεί με τεράστια επιτυχία στην επίλυση αυτού του προβλήματος. Μερικοί εξ' αυτών, ο **Timberwolf** [SS85] είναι ένας από τους πρώτους αλγόριθμους που χρησιμοποιούν την τεχνική *εξομοίωσης ξεπυρώματος* - μιμούμενος βιομηχανικές τεχνικές για να επιτύχει βέλτιστα αποτελέσματα. Επιπλέον ο αλγόριθμος **GORDIAN** [Kle+91] είναι ένας επαναληπτικός αλγόριθμος που αποτελείται από συνεχόμενες επιμειώσεις του μήκους των καλωδίων και τεχνικές διχοτόμησης για να επιτύχει την βέλτιστη τοποθέτηση. Εν τούτοις, οι **Στοχαστικές** αλγόριθμοι υπολείπονται σε δυνατότητα απόλυτα βέλτιστης λύσης (δηλαδή εύρεσης του ελάχιστου μιας σύνθετης αναλυτικής συνάρτησης κόστους) ή αρκούντως γρήγορα (η αναζήτηση στο πλήρες πεδίο των

λύσεων είναι δυνατό, αλλά εξαιρετικά χρονοβόρο)

Σε αυτή την έρευνα, παρουσιάζεται η υλοποίηση, βελτιστοποίηση και κριτική ενός αλγορίθμου **Τοποθέτησης**. Παρουσιάζουμε μία καινοφανή εξέλιξη του κομματιού της **Γενικής Τοποθέτησης** του **Αναλυτικού** Αλγορίθμου **Τοποθέτησης** NTUPlace3 [Che+08]. Ο αλγόριθμος NTUPlace3 επιλέχθηκε λόγω της αξιέπαινης επίδοσής του από άποψη χρόνου, καθώς και της ελευθερίας παραμετροποίησης που παρέχει. Παρ΄ όλα αυτά, ο εν λόγω αλγόριθμος χρησιμοποιεί αναποτελεσματικές στρατηγικές για να φτάσει στο βέλτιστο αποτέλεσμα (εστιάζοντας μόνο στην **Γενική Τοποθέτηση**). Για αυτόν τον λόγο, ο θεμελιώδης αλγόριθμος **Γενικής Τοποθέτησης** έχει επεκταθεί για να υποστηρίζει έναν αλγόριθμο ταχύτερης υπολογιστικά σύγκλισης, με δυναμικά διαμορφωμένο μήκος βήματος, με σκοπό την επιτυχία ταχύτερης εκτέλεσης.

Επιπρόσθετα, επιχειρούμε να ενσωματώσουμε στον αλγόριθμο μια διαφορετική συνάρτηση κόστους, με σκοπό να του δώσουμε την δυνατότητα να μεταχειρίζεται τα στοιχεία, όχι ως μονοδιάστατα στοιχεία σε Καρτεσιανό σύστημα συντεταγμένων, αλλά ως δισδιάστατες μονάδες. Αυτή η αλλαγή ενισχύει το επόμενο βήμα - την διαδικασία **Κανονικοποίησης** με σκοπό να απαλύνει το βάρος της σύνθετης αυτής διαδικασίας.

Εκτελέσαμε τα πειράματά μας και τις συγκρίσεις μας μεταξύ των πλεονεκτημάτων του αλγορίθμου NTUPlace3 και του **aWarePlacement**. Τέλος, ο αλγόριθμός μας ενσωματώθηκε με ένα βιομηχανικό εργαλείο **EDA**, λαμβάνοντας υπ΄ όψιν τις δυνατότητες και τους περιορισμούς του.

Analysis and Implementation of Analytical Placement Algorithms for Microelectronic Circuits

Ntentos Stavros

Submitted for the degree of

Electrical and Computer Engineering

May 2017

Abstract

In **Electronic Design Automation (EDA)** technology nowadays, it is of paramount importance to materialize new and advanced **Integrated Circuits (ICs)** efficiently, automatically, and sometimes with the smallest physical footprint possible. There is a variety of algorithms that are being used to achieve that - but they all serve a common purpose: Give birth to a physical entity through code, and then materialize it. It is common to break a high-level task into smaller ones, and **EDA** is not an exception.

In the first step of design instantiation (also known as **Placement**), the objective is to place the cells in such a way as to minimize various objectives. Simple objectives, such as minimal wirelength, but also complex ones, such as not to allow them to overlap each other at all. After this stage is through, the design of **IC** is prepared for the next steps of the **EDA** process which are **Physical Verification and Signoff**, **Fabrication**, and finally **Packaging and Testing**.

So far, **Combinatorial** methods have been used with great success to solve this problem. To name a few, **Timberwolf** [SS85] is one of the first algorithms to be using the *simulated annealing* technique - mimicking an industrial process to achieve optimal results. Also, the **GORDIAN** [Kle+91] algorithm is an iterative placer consisting of successive wirelength minimization process and partitioning schemes to achieve optimal placement. However, **Combinatorial** algorithms lack the ability

to solve this problem fully optimally (reaching a global minimum on the complex analytical cost function) or fast enough (scanning the entire solution space is a possibility but it is time-consuming).

In this work, implementation, optimization and the evaluation part of a **Placement** algorithm are presented. We present a novel evolution of the **Global Placement** part of the NTUPlace3 **Analytical Placement** algorithm [Che+08]. NTUPlace3 has been chosen due to its great performance in terms of timing and the liberty of parametrization it provides. However, the algorithm in question uses ineffective strategies in order to reach is optimal result (as far as **Global Placement** is concerned). In this way, the fundamental **Global Placement** algorithm has been extended to support a computationally easier convergence algorithm, with finely-tuned step size, in order to achieve faster execution.

Moreover, we attempt to expose the minimization algorithm to a different cost-function, in order to make it aware that the optimization points are not 1D points in a Cartesian coordinate system but rather a 2D unit. This change assists the next step - the **Legalization** process in order to ease the burden of this complex process.

We performed experiments and comparisons between the features of NTUPlace3 and **aWarePlacement**. Finally, the algorithm has been integrated into developing an industrial **EDA** tool, taking its traits and restrictions into account.

Declaration

The work in this thesis is based on research carried out at the Hardware Lab (by **Dr. Sotiriou Christos**), at the University of Thessaly, Volos, Greece. In accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes no part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

SIGNED: ..... DATE: .....

Acknowledgments

I would like to thank my professor, **Dr. Sotiriou Christos** for helping my and guiding me, dozens of hours and the discussions we shared through this journey. I am grateful for **Dr. Stamoulis Georgios**' assistance in the smooth execution of my thesis as well as his wise input on the subject For helping me with the behind-the-scenes mathematics, I would like to thank my professor, **Dr. Nestor Eumorfopoulos** and also **Dr. Anders Skjäl** from Åbo Akademi.

Additionally, I would like to thank all my fellow students and lab mates **Nikolaos Sketopoulos**, **Angelina Delacura**, **Xanthos Vlachos**, **Stavros Simoglou** and **Michalis Giaourtas**, for their support and insight in some of the hardest points in my research. Moreover, I'd like to thank them for the time they invested in me, research or otherwise.

I would like to also thank **my family** for supporting me, both financially and emotionally through all my academic studies, and then some.

Finally, it would be appropriate to thank **Yuliya Avdyusheva** for fully proof-reading my entire thesis and presentation, **Joonas Peltonen** with his suggestions and insights on some of the figures used here and, also, **M. Imran** from University of Durham who prepared the template I used in my Thesis.

Last but not least, I would like to thank *every single person* I might have forgotten. After all, interaction with people shapes one's self.

Contents

Chapter 1

Introduction to EDA

EDA stands for **Electronic Design Automation (EDA)**. **EDA** refers to both a process and a toolset, which is mandatory nowadays for the design and production of **ICs**. Once a process of manual labor and testing, required multiple highly-specialized people on design, with multiple testing, failures, and large to gigantic designs. New consumer and business demands alike nowadays (processing power, smaller energy footprint) and new technologies (smaller physical design, production flow automation) pushed to a change in the design process itself.

It is easier to understand the concept of the phrase, and process itself, if we break it down to words:

Electronic refers to any object that is operating through the use of many small electrical parts (such as microchips and transistors) [Mer17]. Anything electronic, as small as a pacemaker or a calculator, or as big as a space station or an aircraft contain at least one **IC** - simple or complex.

Design is a major factor in any production workflow - and rightfully so! It is the step of the process that genuine ideas are merged together with previous knowledge and experience, that result in a new revolutionary product. That product should be ready to solve the issues we decided upon in the previous step of the process, abiding by all and any restrictions we also carry with us. In specific, in **EDA** it is also the place that the expected behavior is modeled in the **IC**. In some **EDA** designs, it is also part of the process to ensure the correctness and the manufacturability of the design. **Very-Large-Scale Integration (VLSI)** is a sub-

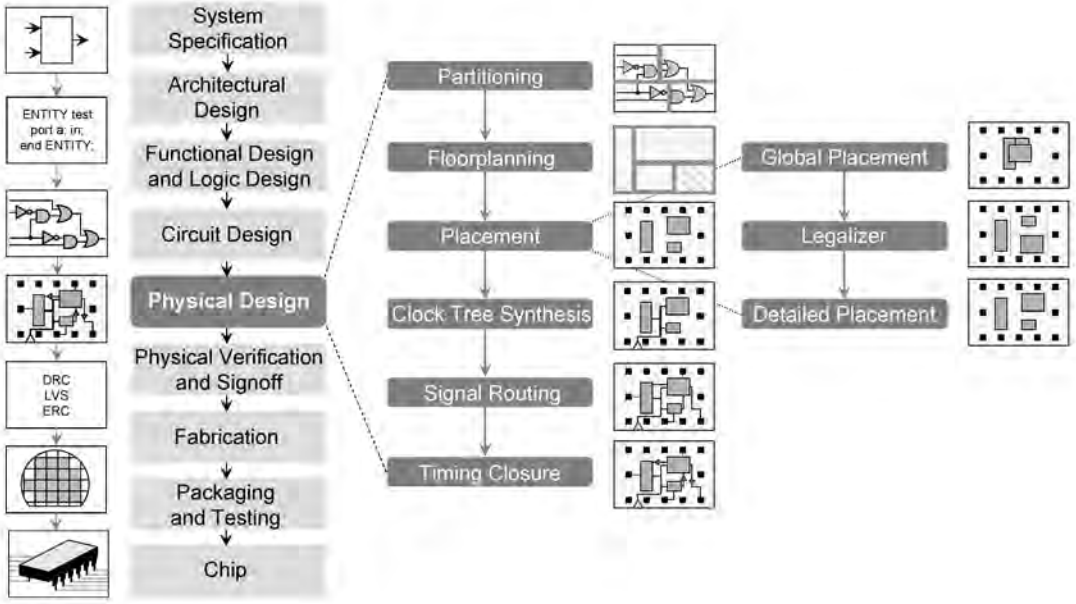

Figure 1.1: Design Flow [Lin12] (adapted to focus on "**Placement**" step)

process of **EDA**, where the **IC** is commonly defined as a fully functional electronic device, occasionally considered as a consumer device. The logic of "build first, fix later" does not easily apply to this process due to the fact that "updating" a printed **IC** is not as straightforward as updating a computer software would be.

Automation is both a blessing and a necessity. Automation helps us cope with the redundancy in every **IC** design, which saves us mindless "designing" on redundant parts. However, the complexities of the designs and the technology nowadays, require an automatic way of handling all the parameters needed for a successful design. So much so that a person (or a team of people) are not able to handle the complexity of the problem. In addition to that, automating processes means that the computer can work (e.g. testing designs), in parallel to human work as well (e.g. refining designs).

EDA has evolved to be a mainstream process - and a composite one indeed, fundamental in creating complex **IC** designs. **EDA** software that has been created aid the designer in producing high-quality designs and ease designer's workflow.

1.1 Physical Design - Placement

IC design consists of many steps, as shown in Figure 1.1. We can roughly split all those steps in three big groups: **Front-end Design**, **Back-end** (or **Physical**) **Design**, **Manufacturing** (or **Fabrication**) **Process**. As a rule of thumb, **Front-end Design** is where the functionality of the **IC** is decided and programmed. **Back-end Design** is where the specifics of the technology come in place and shape the circuit design into what resembles an actual chip. That's why it is also called as "**Physical Design**". Finally, the **Manufacturing** (or **Fabrication**) **Process** is where the actual chip (or, more specifically tens or hundreds of chips) are actually created - otherwise called as "*printed*".

Placement is the first step in **Physical Design** where our circuit begins to exist beyond simple schematics, as opposed to being simply a circuit design. This step is where all of **IC** components are assigned to technology-specific quantums for geometric dimensions (as opposed to simple points) and placed on a 2-dimentional chip area. With the proper materials and manufacturing process, this will ensure properly working components that compromise the **IC**, and thus proper working of the circuit. This representation, which resembles the actual top-view of the printed chip, its called **IC** layout. **Placement** is split into several sub-steps, closely connected to each other - but they all serve the same purpose: optimize multiple objectives at the same time.

Placement is a crucial step in this process, in the sense that, now that the chip is starting to be shaped, any decision made here could decide a variety of issues. A blatantly inferior **Placement** will severely affect any possible optimization. In addition to the automatic cell **Placement** (which is to be carried out in this step), we also have to consider large array macros (hard or soft ones) that need to be placed manually - for example RAM modules. This is done due to the complexity of automating their **Placement**. Deciding each component's **Placement** on the **IC** influences, in turn, interconnection complexity and delays, further affecting the optimization process.

1.1.1 Placer Optimization Parameters

For the most part, a placer is trusted with the assignment of all components, while typically optimizing the following objectives:

- Wirelength: Minimizing the total wirelength in an **IC**'s is a common and really easy decision, considering that:

  1. It minimizes **IC** monetary cost by minimizing wire material
  2. It minimizes the complexity of the design
  3. It minimizes the required energy needed to power the **IC**, and, in turn ...
  4. It minimizes the operating temperature of the chip (thus prolonging its life expectancy) while optimizing cooling costs.

- Signal Delay: The maximum clock cycle of an **IC** is a function of the **Critical path**. The longer the **Critical path** is, the slower clock a said design is able to achieve. Usually, we used to hold accountable the **IC** components for signal delay. However, continuously shrinking the fabrication process made the wires a considerable factor, so much so as to be comparable to component delay.

- Wire Congestion: In our attempt to place cells as close together as possible, there is the possibility that we will fall into the caveat of bundling a lot of cables through the same area. Since cables are created only horizontally and vertically (comparable to **Manhattan Distance** drawing logic), it is even easier to trigger such condition. It is also one of placement requirements to meet the routing resources within all local regions of the chip's core area, so that **Router** can do its job. A congested region might lead to excessive routing detours, or even make it impossible to complete all routes.

- Temperature: In addition to wirelength minimization, it is also possible to distribute the circuit components with high **Switching activity** around the chip (with some limitations). Components draw power (and thus generate heat) whenever they are changing their status low-to-high [Ade14, eq. 17.78, p. 1151]. However, when transistors move high-to-low, the stored electrical energy becomes heat, which, if clustered, could potentially damage the **IC**.

All things considered, we would *prefer* if said objective is achieved in the minimal required time. So, a secondary objective would be *run-time minimization*.

1.1.2 Decomposing Placement Step

Placement itself is split into 3 stages (Fig. 1.2):

- **Global Placement**, is the step where the circuit gets its rough, initial schematic. **Combinatorial**, or usually these days **Analytical** method(s) are trying to optimize the aforementioned optimization parameters, while also removing - as much as possible - all cell overlaps. Wirelength is approximated using one of the various models that exist (more recently the **Half-Perimeter WireLength (HPWL)**). For schematics that include more than 200.000 components, standard cells are grouped in such a way that group-to-group connections are minimized. Afterwards, the clusters (the pseudo-components in place of groups of cells) are minimized in place of their components, then they are minimized instead.

- **Legalization**, is the step that takes the input from **Global Placement**, which are placed in continuous coordinates. Each cell is moved from the position assigned by **Global Placement**, and transforms them in quantitized, discrete coordinates, according to equally-spaced, **Placement** rows and columns. Typically, this is done in such way that the displacement is kept to a minimum, so as to keep all the benefits from **Global Placement**, but also to achieve a printable placement.

- **Detailed Placement**, is the step that takes the input from the **Legalization** step, and there is an attempt to further improve it with regard to a specific objective. **Detailed Placement** techniques include swapping neighboring cells and sliding cells to one side of the row if unused space is available. Both are used to reduce total wirelength. Other possible objectives are optimizing routability, should route topologies can be determined.

As mentioned, **Global Placement** methods are split into **Combinatorial** and **Analytical** algorithms. Starting with the oldest one, **Combinatorial** techniques,

using logic and conditionals, attempt to arrive to an optimal solution. **Combinatorial** logic uses paradigms from real life, and human-like behaviors in an attempt to scan the whole solution space, in an optimized fashion (since the whole solution space would be prohibitively enormous). While easier to come up with, **Combinatorial** algorithms have non-deterministic behavior, so their results cannot be replicated easily or consistently. There are also algorithms that make options based on random input, thus totally unpredictable. Sometimes, they are unable to result in an optimal solution, especially algorithms that are based on random parameters. The upside of that, however, is that it's really hard to arrive at a worst-case scenario, thus eliminating the need to calculate the worst-case complexity. Similarly though, there is no best-case scenario either.

Analytical techniques, on the other hand, use mathematical equations and logic to arrive at an optimal solution. **Analytical** algorithms use mathematical theorems and logic to arrive, deterministically, to the expected solution. Positive aspects of **Analytical** methods include fast execution time and deterministic behavior. Unfortunately, mathematical logic is hard to comprehend, complicated to apply to every solution and sometimes it is not guaranteed to arrive at the minimum solution, due to complexities of the objective function. The main complexity of the objective function would be the way that the local minima are dispersed in the solution space, with regard to the global minimum. The upside is that the algorithm otherwise is predictable and theoretically easier to study. We can visually see the classification of placing algorithms, and some example placers, such as Dragon [YCS03], Kraftwerk++ [VM16], and NTUPlace3 [Che+08].

1.2 Thesis Motivation and Purpose

We have extensively presented the **Placement** problem and it is indeed a serious issue that must be continuously addressed and optimized. After all, this much work from such talented people couldn't have happened in vain. Being an interesting and complex subject, I decided to explore it further in my thesis, which I am presenting before you.

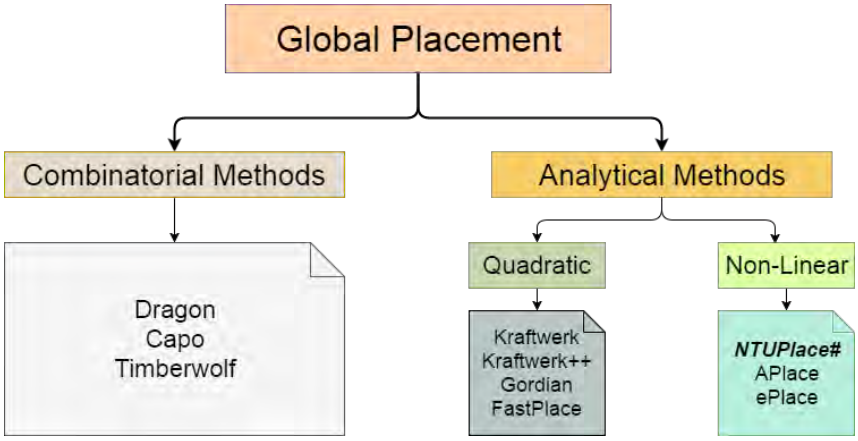

Figure 1.2: Classification of **Placement** Methods

In the following pages, we will explore various scientific fields and previous work done respectively, and afterward, we will explain how we decided to merge all these ideas in one system. We will list the whole developing progress, including all the attempts, libraries, and verification methods used in development. Additionally, we will list all misconceptions and caveats in the development process as well. Apart from the pursuit of knowledge, which is too extensive and complex to be listed here, my thesis objectives were clear:

1. Implement a Generic, Non-Convex **Analytical** Cost Minimizer
   (a) Asses the complexity of a minimizer, for both convex and non-convex functions (from mathematical literature)
   (b) Explore the **Placement** field for prior work (e.g. APlace [KW05], NTUPlace3 [Che+08] etc)
   (c) Implement and verify said Non-Convex Analytical Cost Minimizer
2. Analyze minimizer's operation, quality of results, and enhance it:
   (a) Either by minimizing execution time
   (b) Or by providing better results
      i. Atomically, i.e. better metrics for the **Global Placement** itself
      ii. Or collectively, by providing better results for the **Placement** family
   (c) Or by a balance of the two, i.e.

i. Faster with optimized results “only as much as needed”

ii. Or slower with the best results

We will thoroughly go through the implementation, explaining all caveats and difficulties in our path. We will also extensively test, and discuss regarding said tests and various metrics - regarding both the quality of results and the method’s itself. Usually, the developing mode of this whole thesis can be summarize below:

1. Background Study
2. Improvise
3. Pitch
4. Small Scale Build & Feasibility
5. Full Scale Build
6. An endless loop of: Test, Fix, Verify

This thesis is split into 5 chapters. We began with a general introduction to the problem and their aspects. We will continue with some background into NTUPlace3 **Placement** algorithm and some minimization background. In Chapter 3, we will present our own novel solution, **aWarePlacement**, to deal with these issues. We will also present the results of our implementation. Finally, we will draw our conclusions and present our recommendation for future work regarding the algorithm.

Chapter 2

Implementation Background

In this chapter, we will present the **Analytical Global Placement** algorithm that is the basis of our implementation. Additionally, we will explore the mathematical background regarding the minimization of functions. Finally, we will explore the background of some optimization methods.

2.1 Analytical Global Placement

NTUPlace3 Algorithm is "An Analytical Placer for Large-Scale Mixed-Size Designs with Preplaced Blocks and Density Constraints". Paper [Che+08] explains the solution thoroughly:

i. Uses log-sum-exp wirelength model

ii. Uses clustering to deal with large amount of components

iii. Initializes component placement by solving QP (minimum $W(x, y)$)

iv. Solves the minimization problem using the **Conjugate Gradient (CG)** method for

$$W(x, y) + \lambda_m \sum (\hat{D}_b - M_b)^2, \quad \lambda_m \to (0, \infty) \tag{2.1}$$

v. **Legalization** execution time is utilized in the late stages of **Analytical Global Placement** instead of being a process of its own. It is used to

declare the final accepted solution (even if a future **Analytical Global Placement** gives otherwise better metrics)

vi. **Detailed Placement** techniques are considered following **Analytical Global Placement** iteration to refine the uncoarsening

For the purposes of this thesis, we will only focus on the **Analytical Global Placement** part of this IEEE Transaction paper. As such, we also ignore **Legalizer** and **Detailed Placement** solution and logic.

2.2 Wirelength Model

The main objective of the **Placement** problem is optimizing all parameters mentioned in Subsection [1.1.1]: Minimize wirelength given no overlaps between standard cells. In order to approximate that, mathematically, we have to consider some kind of abstraction between a real Standard Cell and its representation in our mathematical methods; however, this is only the first abstraction that we are going to use. We also need to be able to somehow measure the wirelength between all those points, taking also into account possibly some real world limitations.

2.2.1 Half-Perimeter WireLength (HPWL)

The most commonly used measurement of wirelength is the **HPWL** [Cha08, p. 349, s. 18.2]. The placement input is modeled as a *hypergraph* $G_h = (V_h, E_h)$ with vertices $V_h = \{v_1, v_2, ..., v_{m+p}\}$ representing circuit cells and hyperedges $E_h = \{e_1, e_2, ..., e_n\}$ representing circuit nets. We denote m as movable cells and p as the preplaced elements, $(n, m, p \in \mathbb{N})$.

For any given net $e \in E_h$ the **HPWL** can be expressed as:

$$\text{HPWL}(e) = \max_{i,j \in e, i<j} |x_i - x_j| + \max_{i,j \in e, i<j} |y_i - y_j| \tag{2.2}$$

And thus, the total wirelength of all net (hyperedges E_h) is $\sum_{\forall e \in E_h} \text{HPWL}(e)$. **HPWL** is neither a strictly convex function nor smooth - hence not differentiable. This unfortunate limitation comes from the usage of absolutes in the function

definition and as such cannot be easily minimized. Nevertheless, **HPWL** is a reasonably close approximation to routed wirelength.

In the place of the initial **HPWL** implementation, the log-sum-exp approximation is used instead:

$$\text{HPWL}_{e\in E_h}(e) = \gamma \sum \Bigg(\Big(log \sum_{u_\kappa \in e} exp(x_k/\gamma) \Big) + \Big(log \sum_{u_\kappa \in e} exp(-x_k/\gamma) \Big) + \Big(log \sum_{u_\kappa \in e} exp(y_k/\gamma) \Big) + \Big(log \sum_{u_\kappa \in e} exp(-y_k/\gamma) \Big) \Bigg) \tag{2.3}$$

Let us break down the equation. First of all, for input e, the expression

$$log \sum_{u_\kappa \in e} exp(x_k) + log \sum_{u_\kappa \in e} exp(-x_k), u_k : (x_k, y_k) \text{ approximates } \max_{i,j\in e, i<j} |x_i - x_j|$$

This expression has all the properties we want to streamline its minimization, however, it is subject to arithmetic underflow/overflow. So, we introduce a normalization factor γ comparable to the input values.

$$\gamma \sum_{\forall e \in E_h} \Big(log \sum_{u_\kappa \in e} exp(x_k/\gamma) + log \sum_{u_\kappa \in e} exp(-x_k/\gamma) \Big), \quad u_k : (x_k, y_k) \tag{2.4}$$

Likewise, we append y values to calculate the expression. According to the authors of [Che+08], γ values that are 1% of chip width ("reasonably small") remedy *both* arithmetic issues *and* provide an accurate representation of Equation 2.3. For reasons we will explain further, the implementation we will use is the simple **HPWL** implementation.

2.3 Density Function

For reasons comparable to the **HPWL** transformation (convexability, systemic minimization), authors decided to express the density function D_b

$$D_b = \sum_{\forall v \in V} P_x(b, v) P_y(b, v) \tag{2.5}$$

where $P_{\#}$ is the overlap function for b : bin and v : block in both $x-, y-$ directions (unified under hash (#) symbol). However, since the overlap function is non-smooth, authors adopt the bell-shaped function $p_{\#}$ in place of $P_{\#}$, defined as:

$$p_{\#}(b,v) = \begin{cases} 1 - a{d_{\#}}^2, & 0 \leq d_{\#} \leq \frac{w_v}{2} + w_b \\ b(d_{\#} - \frac{w_v}{2} - 2w_b)^2, & \frac{w_v}{2} + w_b \leq d_{\#} \leq \frac{w_v}{2} + 2w_b \\ 0, & \frac{w_v}{2} + 2w_b \leq d_{\#} \end{cases} \tag{2.6}$$

$$\text{where} \begin{cases} a & = \frac{4}{(w_v+2w_b)(w_v+4w_b)}, \\ b & = \frac{2}{w_b(w_v+4w_b)}, \\ w_{\#} & \text{width } (b: \text{bin and } v: \text{block}), \\ d_{\#} & \text{center-to-center distance in \# direction} \end{cases}$$

Using c_v as a normalization factor (to match the block potential and its area), final density function is:

$$\hat{D}_b = \sum_{\forall v \in V} c_v p_x(b,v) p_y(b,v) \tag{2.7}$$

2.4 Analytical Global Placement Algorithmic Properties

2.4.1 Minimization of a piece-wise non-convex function

The NTUPlace3 cost function, shown at the beginning of this chapter as Equation [2.1], compromises of two parts: First, we have the wirelength part and then we have the density part. Wirelength is a convex function and density function presented, is a non-convex one; hence the sum is a non-convex function. In order to solve this non-convex function **Analytically**, the authors decided to first solve the convex part of the equation, and then proceed with the non-convex part. By doing so, the authors gain a significant advantage: They have a somewhat-optimal solution, and

then proceed to further optimize it. Just what most of us try to do in real life.

Non-convex functions have a significant drawback: By computing their entire solution domain, we can see that there is no direct correlation between the global minimum of the function and its computed derivative. Derivatives can help to pinpoint (and then block any subsequent optimization) to local minima only. Advanced methods are required in order to quickly arrive at the global minima - that is, if the minimizer would actually be able to do so.

In order to circumvent such issue, authors of the paper mimicked a human approach in the algorithm: What do we usually do when we have to satisfy a complex task with increased coupling over its dependents? We start by satisfying known / easy tasks, and then move forward to most complex ones, attempting to overlap all individual steps. So this is what they did: They started by solving the easily solvable issue of minimal wirelength, by solving the Quadratic **Placement** problem. In further iterations, bit by bit, they introduce an increasing dependency to the density minimization function that is attached to the density of the placed components in a given bin. This weight factor is computed by the mathematical expression in Equation [2.8].

$$\lambda_0 = \frac{\sum_{\forall e \in E_h} |\partial W(e)|}{\sum_{\forall b \in Bins} |\partial \hat{D}_b|}, \qquad \lambda_{i+1} = 2\lambda_i, i \in \mathbb{N} \tag{2.8}$$

2.4.2 Overflow Ratio

A lot of algorithms decisions are taken on the basis of a metric, that authors call $overflow\_ratio$. $overflow\_ratio$ is defined as

$$overflow\_ratio = \frac{\sum_{\forall b \in Bins} max(D_b - M_b, 0)}{\sum \text{total movable area}} \tag{2.9}$$

$overflow\_ratio$ is explained as an overall chip metric, as opposed to a more localized approach. Intuitively, this metric shows the % available moving space across the **IC**. Theoretically, this metric shouldn't be negative, otherwise, the **Placement** would be unable to happen (less available space than the sum of the component area).

2.5 Conjugate Gradient Search with Dynamic Step Size

The NTUPlace3 Algorithm, instead of simply applying the proposed gradient solution, attempts to take it one step further. It is possible that minimization towards the chosen current direction could be much bigger, thus arriving at the expected solution much faster. Similarly, proposed direction may need to be applied to a minimized extend, and then take a different direction.

Their predecessor, APlace [KW05], is using Golden Section Line Search to find the proposed step size. NTUPlace3 authors claim that Golden Section Line Search algorithm takes most of the optimization time, hence delaying the process more than needed. In its place, they propose "a more efficient method". Their step size is calculated through:

$$\alpha_k = \frac{s \cdot w_b}{\|d_k\|_2} \tag{2.10}$$

where s is a user-specified scaling factor, whereas w_b is the bin width. Their selection is defended by the fact that it limits the spreading of blocks, since the total quadratic Euclidean movement is fixed as

$$\sum_{v_i \in V_h} (\Delta x_i^2 + \Delta y_i^2) = \|\alpha_k d_k\|_2^2 = s^2 w_b^2 \tag{2.11}$$

where Δx_i and Δy_i are the amount of movement along the #-direction for the block v_i in each iteration. s user-specified scaling factor determines the precision of the returned solution. Smaller s values lead to better results; however, increases run time and vice-versa. The authors presented a complex figure of s's correlation with CPU time and **HPWL** metric and concluded that the optimal point lies in $[0.2, 0.3]$ bracket.

2.6 Review of Analytical Global Placement methods

After an extensive reviewing of the available **Analytical Global Placement** methods, it is clear that mathematics, play a major role in hardware placement. In general, mathematics is deeply embedded into software, in encryption and formal verification methods, and as we see now, in hardware as well. What is more important is that mathematics are greatly benefited from computer engineering, as well. Representing problems in the real world, especially in the brink of the continuous and discrete world is not achievable without its own set of problems.

However, it is of great significance, and also mandatory to be done **Analytically** to ensure predictability in execution. Apart from the already sped up execution, we are also able to introduce additional mathematical methods to further accelerate execution, as such issues have already been solved from a mathematical aspect. We just point and click!

2.7 Armijo-Goldstein Backtracking Line Search

Given a starting position $\mathbf{x}$ and a search direction $\vec{\boldsymbol{p}}$, the task of a line search is to determine a step size α that adequately reduces the objective function $f : \mathbb{R}^n \to \mathbb{R}$ (assumed smooth), i.e., to find a value of α that reduces $f(\mathbf{x} + \alpha\,\mathbf{p})$ relative to $f(\mathbf{x})$. However, it is usually undesirable to devote substantial resources to finding a value of α to precisely minimize f. This is because the computing resources needed to find a more precise minimum along one particular direction could instead be employed to identify a better search direction. Once an improved starting point has been identified by the line search, another subsequent line search will ordinarily be performed in a new direction. The goal, then, is just to identify a value of α that provides a reasonable amount of improvement in the objective function, rather than to find the actual minimizing value of α.

2.7.1 Algorithm Implementation

The backtracking line search starts with a large estimate of α and iteratively shrinks it. The shrinking continues until a value is found that is small enough to provide a decrease in the objective function that adequately matches the decrease that is expected to be achieved, based on the local function gradient $\nabla f(\mathbf{x})$.

Based on a control parameter $\boldsymbol{c}$, which lies within the $(0, 1)$ interval, the Armijo-Goldstein condition tests whether a stepwise movement from the current point $\boldsymbol{x}$, to $(\boldsymbol{x} + \boldsymbol{\alpha p})$ achieves a significant decrease in the objective function, where $\boldsymbol{\alpha}$ is the step and is to be determined, and $\vec{\boldsymbol{p}}$ is the vector direction which presents some decrease, and is typically a unit vector. The Armijo-Goldstein condition is satisfied if the following inequality holds.

$$\boldsymbol{f(x + \alpha p) \leq f(x) + \alpha c m} \tag{2.12}$$

Equation 1: Armijo-Goldstein Condition Inequality

This condition, when used appropriately as part of a line search, can ensure that the step size is not excessively large. However, this condition is not sufficient on its own to ensure that the step size is nearly optimal. Any value of α that is sufficiently small will satisfy the condition.

Thus, the backtracking line search strategy starts with a relatively large step size, and repeatedly shrinks it by a factor $\tau \in (0, 1)$ until the Armijo-Goldstein condition is fulfilled. The search will terminate after a finite number of steps for any positive values of c and τ that are less than 1. For example, Armijo used $\frac{1}{2}$ for both c and τ in a paper he published in 1966.

Based on this condition, an algorithm may be devised, presented in [Wik17a], which incrementally computes the step, until the above condition is indeed satisfied. The outcome of the algorithm will be a new position $\boldsymbol{x' = (x + \alpha p)}$, in the direction of the specified gradient $\vec{\boldsymbol{p}}$.

Algorithm 2.7.1: Armijo-Goldstein backtracking line search algorithm

Data: Maximum Candidate Step Size Value: $\alpha_0 > 0$
Search Control Parameters: $\tau \in (0,1)$ and $c \in (0,1)$
Minimization Function: $f(x)$
Result: α_j as solution

```
t = −cm;
j = 0;
while f(x) − f(x + α_j p) < α_j t do
    Increment j;
    Set α_j = τ α_{j−1};
end while
/* In other words, reduce α_0 by a factor of τ in each iteration
   until the Armijo-Goldstein condition is fulfilled          */
```

2.7.2 Algorithm Pseudo-code

Starting with a maximum candidate step size value $\alpha_0 > 0$, using search control parameters $\tau \in (0,1)$ and $c \in (0,1)$, the backtracking line search algorithm can be expressed as follows:

1. Set $t = -cm$ and iteration counter $j = 0$.

2. Until the condition is satisfied that $f(\mathbf{x}) - f(\mathbf{x} + \alpha_j \mathbf{p}) \geq \alpha_j t$, repeatedly increment j and set $\alpha_j = \tau \alpha_{j-1}$.

3. Return α_j as the solution.

2.7.3 Usage of Armijo-Goldstein

Usage of Armijo-Goldstein is easier shown that explained in writing for the rest of us. Let random function $f(x)$ that we want to minimize (shown annotated at Figure [2.1]). If we were to use any conventional **CG** minimizer, we would make the movements shown by the orange arrow, and then we would move shown by the purple arrow on the function plane. While minimizer would be minimized, but that wouldn't reach the best available option.

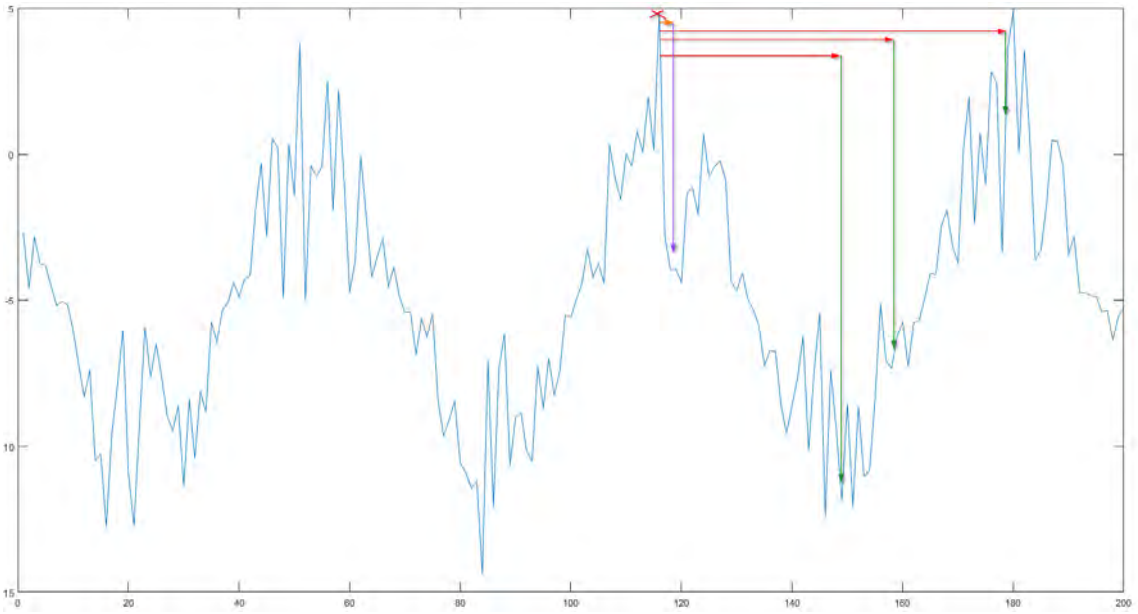

Figure 2.1: Random function, with minimization annotations

Alternatively, we would like to be able to do the *3rd red/green* option when minimizing. The issue is that when minimizing towards a direction, $d\vec{x}$ is a predetermined vector. What Armijo-Goldstein does is, considers the minimization vector as a unit vector, and uses a scaling methodology: starts off with a big step size, that iteratively scales it down until the Equation [2.7.1] is satisfied. This is a fancy way of questioning whether minimization if indeed minimizes the function, is actually a worthy one.

2.8 Thesis Motivation and Purpose

Concluding our analysis, we can see that NTUPlace3 is indeed an interesting algorithm for **Analytical Analytical Global Placement**. Traits that make it appealing to further improvements are fast execution times, quality of results, and the ability to be parametrized in a handful of aspects.

In addition to the degrees of freedom given by the algorithm, we also considered various other ways to further optimize the solution. We implement and extend the algorithm by using a more mathematical approach to the **CG** multidimensional minimization, and the dynamic step-size control.

Chapter 3

aWarePlacement Implementation and Analysis

In this chapter, a detailed analysis of the implementation of our **aWarePlacement** algorithm is presented - which is the main concept of this thesis. We will also go through a detailed technical analysis of every step of the algorithm. Furthermore, we will report the implementation and the difficulties - along with their resolutions.

Someone would expect that an analytical algorithm, like NTUPlace3 [Che+08], which contains easy-to-understand mathematical techniques is easy to implement. However, this is not always the case - and this case is not different. Even with some mathematical background, which would be necessary / recommended, there are real challenges in implementation. Thus, it was decided to split the developing process in two parts. Proof-of-concept and integration to the industrial **EDA** tool.

Initially, development was done using a separate code base, with a really small example. It was deemed that this would help focus on the implementation itself, rather than being disturbed from the complexities of a fully-featured **EDA** tool. That also helped to decouple the solution from the actual implementation platform, thus making it more modular. Afterward, moving from proof-of-concept to actual implementation was not without facing a fair share of troubles as well. Nevertheless, the coding effort was mostly straightforward after being acquainted with the tool's coding logic and data structures. All coding attempts (unless otherwise specified) are completed in C89 language.

3.1 Net Model

In order to approximate the **IC**'s wirelength mathematically, we consider the Point-to-Point Net Model: every connectable object is represented by a point in the X,Y Cartesian Axes, hence the name of the model. Both Standard Cells and I/O Pins are approximated as points, regardless of the fact that Standard Cells actually have 2-dimensional representation in the physical world. Depending on the implementation, a point would be either in the center or on the top left of the actual cell – or anywhere for that matter. In our implementation, that point can be considered to be both in the center and on the top left of the actual cell, something we can actually select beforehand.

Usually, from the Point-to-Point model, only Cell-to-Cell and Cell-to-Pin connections are actually seen in usage. This is due to the fact that Pin-to-Pin connections make the **IC** work slower; however such connections - in theory - do exist. For instance, in hierarchical design, sub-circuits communicate with each other by connecting their Input/Output pins to provide data to each other. This connection, however, it is not strictly considered Pin-to-Pin, since "Pins" connect an **IC** to the outer world, whereas, both sub-circuits are part of the same **IC** - and thus are not considered "outer".

Connections between Cell-to-Cell and Cell-to-Pin elements are approximated with a line connecting to using **Manhattan Distance** logic. This is because of limitations on the routing stage: Connections inside an **IC** can only exist as a sum of vertical and horizontal lines - never diagonal.

3.1.1 Wirelength Model

In comparison with our previous chapter, we should also reference our wirelength model of choice. Our minimization cost function minimizes the Square Euclidean Distance, which also succeeds in minimizing the **HPWL** cost. In our implementation, we use the original **HPWL** approximation (i.e. using absolutes and max function), but we are not using this metric directly for the minimization. While we discussed the log-sum-exp **HPWL** has some advantages over the original

HPWL model, since we are not using this metric directly for minimization, we are not interested in the properties that this approximation provides us.

3.2 Initial Attempts

We started to build the algorithm - logic and data structures - for both my algorithm and other, closely related methods as well. As soon as we started looking at the input data, we decided it was not going to be appropriate for the mathematical representation of our problem's data. It is required that we hold in memory a matrix that contains the entirety of Cell-to-Cell connections, along with connection logic for Cell-to-Pin components. While our **EDA** tool covers on that end pretty neatly, Cell-to-Cell connections need to be in a more appropriate form. We have already mentioned that placement input is given as a *hypergraph* $G_h = (V_h, E_h)$, and such it is appropriate to be illustrated using the **Laplacian Matrix**.

3.2.1 GNU Scientific Library

For the first stage of my building, we decided to go for the GNU Scientific Library (or GSL for short). This option offered us a kick-start on tackling some mundane tasks regarding vector/matrix handling operations, without much hassle on our side. We also did not have to a) write, and most importantly b) validate the written code for either correctness or performance. It is a common practice to rely on code other people wrote to do what you are trying to achieve. This helps by not wasting man-hours in conception, coding, and verification of functions and algorithms handling your data - let alone a fully featured API. Moreover, we can argue that, for some functions, it is required that you also have a diverse academic background, for example, superb command of mathematical ways to solve problems efficiently. It is not realistic for everyone to be on that advanced level, but if one writes good code, then it is not expected to have extensive knowledge about everything.

Laplacian Matrix is a good option that provides all the needed information to redraw the graph to its full extent, or more importantly, being able to solve it mathematically. However, all is not good yet: due to the nature of the input

data and the **Laplacian Matrix** representation, we will soon end up with matrices unable to be represented in memory. **Laplacian Matrix** requires $O(n^2)$ storage, as it is a $n \times n$ matrix. While many circuits exist with no more than one thousand (1.000) components, which does not require a big amount of RAM to be stored in (8 MB for a circuit that has exactly 1.024 components). However, 1.000 components is sometimes a laughable number of components in our days for real **IC**, for example an FFT **IC** has 32.281 components. Storing the **Laplacian Matrix** for this one requires, more or less 7,76 GB of memory! Maybe that is not a limit for our current technology, as we can have way more RAM than that available, especially using virtualization, but there are also much bigger circuits - sometimes going up as much as one million (1.000.000) components. So this is clearly not the way to go.

3.2.2 Sparse Matrices

If you remember the definition of the **Laplacian Matrix**, and the specifics of a circuit, that matrix is bound to be filled with zeros. That means, we are spending so much memory and resources to actually map ... zeros. It's not a small feat: even in a simple adder circuit, that has 17 components, 224 out of 289 (17^2) matrix cells will be empty. In other words, we are wasting 77.5 % of our allocated memory - which is bound to become even more inefficient as time goes on. Enter sparse matrices: the most efficient way to store matrices that contain a lot of zero elements. There are various methods to achieve that, but we are going to focus on a few to give you the idea what is it about. What all of them have in common, is that if something is not there, it is considered to be zero.

One common form of storing entries is a Coordinate List. That includes storing a list of tuples that contain the (row, column, value) triplet. This form facilitates incremental matrix construction. Ideally if kept sorted, it would decrease random access time. Adjacent to this method is Dictionary of Keys, that uses a dictionary to map (row, column) pairs to the value of the elements - it lacks however in iterating the structure in proper order. Also, closely related is the List of Lists method, that keeps a list of rows and (column, value) pairs in a list. For sorted lists, this method offers fast lookup access.

On the other hand, methods that facilitate arithmetic operations, column slicing, and matrix-vector products would be Compressed Row Storage and Compressed Column Storage. These two methods are closely related to each other, as they operate using the same logic. Their only difference is that, respectively, one compresses rows and the other one compresses columns. They also rotate over the matrix in the same direction (i.e. row-wise or column-wise). Finally, there are other kinds of sparse matrices that solve specific matrices. Examples are Banded (storing band matrices), Diagonal (storing diagonal matrices), and Symmetric (stored as adjacency list since they are derived from adjacency matrices). [Wik17f]

In our **EDA** tool, we use Coordinate List and Compressed Column Storage, interchangeably depending on algorithmic and implementation needs. We can use interchangeably Intel Math Kernel Library and CXSparse libraries for sparse matrices and operations. We also are able to use GNU Scientific Library for simple circuits. In my implementation, I used GNU Scientific Library as a first building block but quickly amended the code to extract data from sparse matrices. In my function, sparse matrices are in Coordinate List form.

3.3 Minimization (or Maximization) of Functions

Minimizing (or Maximizing) of a (cost) function is one of the oldest problems, and one problem we are actually taught in school to solve in various ways, usually using a graph of the quadratic function: $x^2 + ax + b$. To begin with, let us describe the process to mathematically discover a function minimum, which is not so different from finding the root of a single-dimension function. Also, for the simplicity of this text, we will only reference to this method solely as minimization. However, conceptually, we could have referenced the maximization as well, as these processes look very much alike.

3.3.1 Bracketing a Minimum

"Bracketing" could be otherwise described as "finding the area" where (a) minimum lies. We will draw a parallel from bracketing the root. As we learned in school, a

smooth and differentiable's function root is bracketed by a pair of points a, b, if the $f(a)$ and $f(b)$ results bear opposite signs ($+/-$ or $-/+$). We could also say that $a < b$, but that constraint is rudimentary because the same holds true for $a > b$.

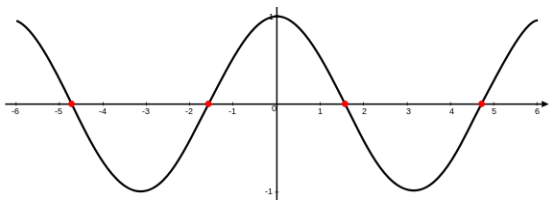

Figure 3.1: Roots of cos(x) function [Pbr08]

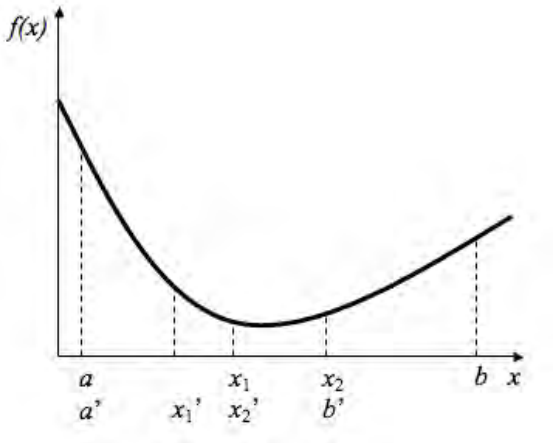

Figure 3.2: Function with single minima

We can understand why if the $f(a)$ and $f(b)$ results bear opposite signs, then a and b bracket the solution by looking at Figure 3.1. By selecting two opposite points around any red point, we can deduce that $f(a)$ and $f(b)$ will always have opposite signs if they bracket any of the roots of $cos(x)$ function. Note that this constraint only tells us that iff $f(a)$ and $f(b)$ results bear opposite signs ($+/-$ or $-/+$), then there is at least one function root laying in-between. It could be that a root lies between $f(a)$ and $f(b)$ results that bear same signs ($+/+$ or $+/+$), for example, in x^2. Along the same lines, multiple minimums can lie between results that bear opposite signs ($+/-$ or $-/+$), which we can verify by picking $|a|, |b| > 2$ and opposite signed in Figure [3.1].

Minimum can be a little more complex: a smooth and differentiable's function minima is bracketed by a triplet of points $a < b < c$ (likewise for $a > b > c$), if the $f(b)$ result is less than both $f(a)$, $f(c)$ results ($f(b) < f(a), f(c)$). This is actually the same condition as the one we have for bracketing function roots. Looking at Figure 3.2, we can see that, conceptually, those two algorithms look-alike. However, since there is no lower point than the function minimum, we cannot test for result signage - this is why we require 3 points instead of two.

3.3.2 Conjugate Gradient Method in Multidimensions

We have described, in a nutshell, how bracketing a minimum in one-dimensional functions works. In order to actually find a minimum, we continue to shrink the bracketing area, until "reasonably" small. It is hard to pinpoint the exact number, as sometimes the discrete nature of floating-point arithmetic (and any kind of computer arithmetic for that matter) makes that impossible to uniquely identify such point. We could argue right here that we have a good rough estimate (and an algorithm) of one-dimensional minimization. We could delve into more detail about bracketing patterns and methods, but it is not necessary at this point in order to move up to multidimensional minimization.

Our first guess would be to consider the exact same logic we used on one-dimensional minimization, only we will have to do it N times, where N is the number of dimensions in our function. While this method works quite well and really intuitively for the test cases we have in our head ($ax^2 + bx + c$ with $a > 1$, and $\frac{x^2}{a^2} + \frac{y^2}{a^2}$ with $a < 1$), it doesn't work quite well in long, narrow valleys (i.e. the reverse from the described test cases). In the worse cases, what we will do, instead of targeting the minimum directly, we will take small steps moving sideways towards the center of the valley. We are always moving in small steps towards each dimension's minima, until some point we actually reach that, instead of targeting directly for the valley.

A much smarter approach would be to compute, instead of just computing the value of a function $f(\mathbf{p})$ at an arbitrary point $\mathbf{p}$, also compute the vector of first (partial) derivatives, also known as gradient $\nabla f(\mathbf{p})$.

Assuming that we approximate the function f as a quadratic function

$$f(\mathbf{x}) \approx \mathbf{c} - \mathbf{b} \cdot \mathbf{x} + \frac{\mathbf{1}}{\mathbf{2}} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{3.1}$$

Then the number of parameters in f function equal the number of independent parameters in $\mathbf{A}$ and $\mathbf{b}$, which is $\frac{1}{2}N(N+1)$. Therefore, we expect to *find* the function's minimum to be in $\Omega(N^2)$ iterations. Calculating function's gradient, however, brings us N new information quantums. Used wisely, we could instead

make only N line minimizations in order to reach the function minima.

We could argue here that there is nothing to be gained actually: N^2 line minimizations are needed in the intuitive minimization method, N line minimizations and N function gradients (vector of N first derivatives) are needed. We can more or less assume (initially) that each first derivative requires about the same time as a function computation. That would mean also N^2 iterations, and as such comparable time.

While we are not achieving optimization of N order, we have to take into account:

1. Each vector component will save, more than a function evaluation, also all the extra costs incurred by initiating a new line minimization.

2. There is often a high degree of redundancy in the formulas for the various components of a function's gradient. When this is so, especially when there is also redundancy with the calculation of the function, the calculation of the gradient may actually cost significantly less than N function evaluations.

Before we rush into implementation, however, we must not make hasty decisions. Methods that utilize gradient information are not "equally good" amongst themselves. Intuitively, probably we would arrive at the *Steepest Descend* method: "From arbitrary point $\boldsymbol{P}_0$, move to the point $\boldsymbol{P}_{i+1}$ along the direction of $-\nabla f(\boldsymbol{P}_i)$". Unfortunately, while more optimal than using no gradients at all, still fails to efficiently solve long narrow, otherwise perfect quadratic valleys.

We are looking for a way that will allow as to minimize the function, not towards the gradient vector, but somehow *conjugate* to the gradient and all previous methods, insofar as possible. We can easily deduct that methods that accomplish this construction are called *conjugate gradient* methods.

Starting from an arbitary g_0, which usually is $g_0 = \nabla f(\boldsymbol{P}_0)$ for arbitary P_0 (in our case, usually $(0,0) \forall$ cell) we set $h_0 = g_0$ and then:

$$\vec{g}_{i+1} = \vec{g}_i + \lambda A \cdot \vec{h}_i, \text{ where } \lambda_i = \frac{\vec{g}_i \cdot \vec{g}_i}{\vec{h}_i \cdot A \cdot \vec{h}_i} \tag{3.2}$$

$$\vec{h}_{i+1} = g_{i+1} + \gamma_i \vec{h}_i, \text{ where } \gamma_i = \frac{(\vec{g}_{i+1} - \vec{g}_i) \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i} \tag{3.3}$$

which, in turn, satisfy the orthogonality and conjugacy conditions:

$$\vec{g}_i \cdot \vec{g}_j = 0, \qquad \vec{h}_i \cdot A \cdot \vec{h}_j = 0, \qquad \vec{g}_i \cdot \vec{h}_j = 0, \qquad \forall j < i \tag{3.4}$$

The algorithm described so far is the original Fletcher-Reeves version of the conjugate gradient algorithm. Later, Polak and Ribiere introduced one tiny, but sometimes significant, change. They proposed using the form

$$\gamma_i = \frac{(\vec{g}_{i+1} - \vec{g}_i) \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i} \tag{3.5}$$

instead of Equation 3.3. While both γ_i definitions are equal by orthogonality conditions 3.4 for exact quadratic forms, usually our function will not be a quadratic form. Even if we arrive at the minimum of the *approximated* quadratic form, we may still need to proceed to another set of iterations. "There is some evidence that the Polak-Ribiere formula accomplishes the transition to further iterations more gracefully: when it runs out of steam, it tends to reset $\vec{\boldsymbol{h}}$ to be down the local gradient, which is equivalent to beginning the conjugate gradient procedure anew" [Pre+07].

3.3.3 Validation Methodology

Non-convex Analytical Minimizer

In our initial approach, I decided to use a simple function for testing: $x^2 + y^2$. This enabled me to verify that my novel implementation of literature minimization references was indeed successful. After that, before I moved on to implement the solution on an industrial **EDA** tool, I wanted to test more complex functions. One of those functions, is the Beale's Function [JY13].

$$f(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2 \tag{3.6}$$

There are many functions to select from, testing various minimization parameters. I selected the one that is closer to test our worst-case scenario: A long

valley, that would otherwise delay / inhibit minimization. It was also the logical step, moving up from a simple two-dimensional function, to a more complex two-dimensional function. It would also facilitate to solve out some logic bugs in the implementation, before moving on to actual **IC** designs.

It should be noted, that all the provided minimization functions used for verification are convex (or concave) functions. The mathematical complexity involved in verifying so early such complicated methods would defeat the purpose of this verification. It is paramount to verify the capability of the minimizer to perform its own job, rather than having to deal with the complexities of any given benchmarking non-convex function.

Minimized Cost Function

Of course, we cannot stop our validation simply at the minimizer. We also have to extend our tests to the correctness of the cost function in question, preferably optimize it - as well as the function's derivative computation. This was done in accordance with our previous methodology (e.g. manual verification). However, at this point, we decided to also utilize the power of Matlab's **M**atlab **EX**ecutable (MEX for short) files. An MEX-file is a way to execute specially-written C functions as if they were functions written in Matlab.

The function we attempt to minimize is the squared Euclidean distance:

$$\sum_{i=1,j=1}^{i,j\in\mathbb{N}} c(i,j)((x_i - x_j)^2 + (y_i - y_j)^2 \tag{3.7}$$

and $c(i,j)$ is declared as:

$$c(i,j) = \begin{cases} 1 \cdot \$\{ConnectionCost\}, & \text{if } c_i, c_j \text{ are connected} \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

The terms $(x_i - x_j)^2$ and $(y_i - y_j)^2$ respectively give the squared horizontal and vertical distances between the selected of i and j. This formulation implicitly decomposes all nets into two-pin subnets. The quadratic form emphasizes the

minimization of long connections, which tend to have negative impacts on timing.

Formulating a simple version of our problem, compiling it with the help of Matlab and Microsoft Visual Studio gave us some interesting results that helped us uncover some important issues with the implementation. One of the most important problems lied in the mathematical representation of the data structures. When solving the **Placement** problem with the Quadratic **Placement** method [Kah+11, p. 110], then the matrix (both the definition and visually) look very much alike the **Laplacian Matrix**. We wanted to respect the convention that code should be split into functions and that each function's execution should result in unique results and not overlap with functions or even partial code.

In our initial approach, we were using a slightly (or largely, depending on the **IC** size) different version of the **Laplacian Matrix**. However, since the matrix looked alike with the **Laplacian Matrix**, the function managed to arrive results that looked (both visually and in metrics) not quite there. It took the combined effort of manpower, re-reading the mathematical formulation and extensive testing methodologies to uncover the implementation flaw, and quickly amend it.

The Matlab suite also enabled us to test the cost function (and it's derivative) using tested, verified, high quality, and a variety of minimizers. The derivative, due to the high complexity of the cost function, resulted in being a numerical one (using the limit theorem) instead of an **Analytical** one. This approach enabled to generate the derivative, regardless of the implemented function - hence decreasing the complexity of the solution by far.

3.4 Armijo-Goldstein Line Search

3.4.1 Implementation - Free parameters: c and τ

We could argue that implementation of the algorithm presented in Section [2.7] is an easy-to-implement algorithm and it's too straightforward to have any degree of freedom. We have to select, however, the algorithm's parameters: c and τ.

Initially, by looking at the algorithm implementation, completed code and small examples, we could argue that we would like to use a small c value, in order to

push the expression $\alpha \cdot t$ to a small value, thus making the algorithm more strict in solution accepting. For the same reasons, we would select a low-to-medium τ value: that would further decrease the α parameter, thus making the algorithm even more strict in solution accepting.

3.5 Current Completed Work

So far, this is what I have completed:

- void loHiPassFilter: A filtering function that simply pushes outlying cells back inside the core area
- void advancedCellLoHiPassFilter: An advanced filtering function that moves all cells towards the center by the max violation amount, instead of simply pushing one inside. Helps to keep the layout of the cells consistent with their placement instead of stacking the outliers with themselves and/or the cells that lie in the perimeter.
- double NTUPlace3: The main minimizer function. The minimizer is programmed according to Armijo-Goldstein CG proposed function (as per 3.3.2 [Pre+07])
- void set/getCellsXY: setter / getter function to communicate with the tool that the code is implemented into
- double quadraticCostFunction: the quadratic minimization method, expressed in a mathematical function. The function is also optimized **Analytically** (*i.e.* factoring the matrix multiplication with the vectors); however, this is not used as current sparse matrix implementation does not allow for such optimizations. The sparse matrix, although stored in ascending i, j key order, is currently an undefined behavior and as such factorization is not able to be used. Computes

$$\sum_{\forall i,j \in cells}^{n} c(i,j)\Big((x_i - x_j)^2 + (y_i - y_j)^2\Big),\ c(i,j) = \begin{cases} c,\ i,j \text{ connection cost} \\ 0, \text{ if } i,j \text{ unconnected} \end{cases} \tag{3.9}$$

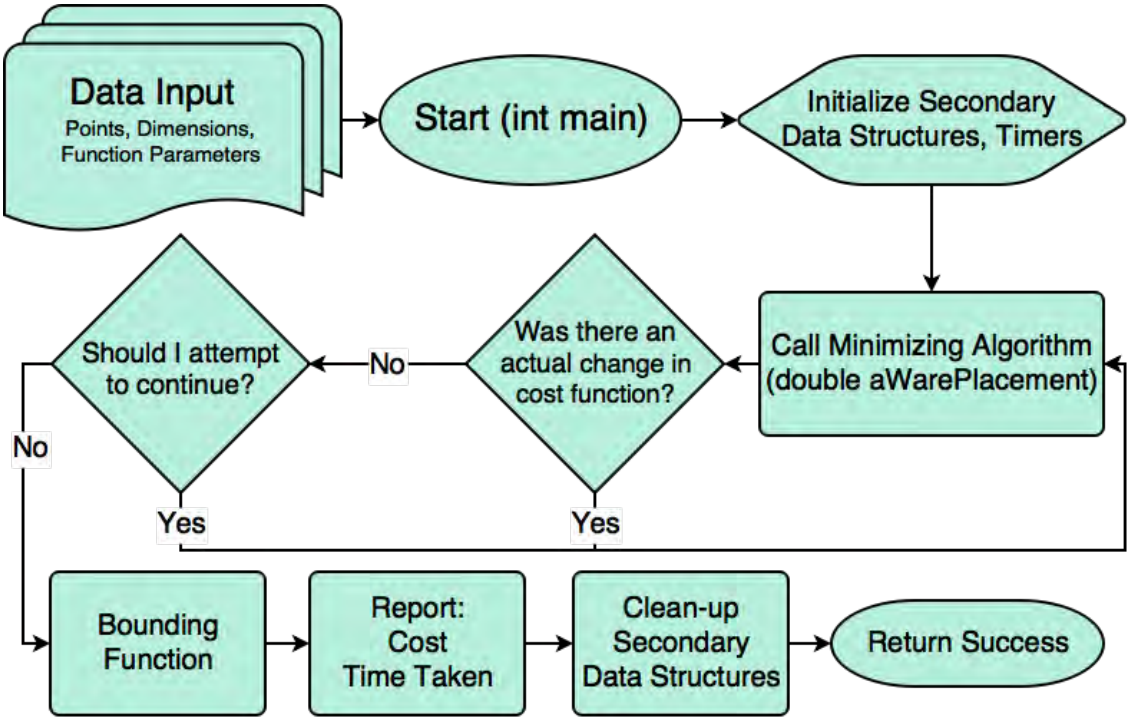

Figure 3.3: High-Level Meta-algorithm

and returns the Squared Euclidean Distance as cost.

- void quadraticCostFunctionDF: the derivative of the function is currently computed arithmetically. It is possible that analytical computation is achievable but, further down the road where density function's arithmetic derivative is used, it is probably not gonna make a difference. h-factor is set to half of the chip's respective quantum dimension (so that it may jump back and forth at maximum by one row/column).

- int main_NTUPlace3Minimization: Preparatory function for all the previously mentioned functions

Two flow charts describing the execution path are presented in Figures [3.3, 3.4].

3.6 Implementation Notes

Connecting all steps of the algorithm flowcharts to appropriate sections is no hard task - but we would like to shed some light on "Compute" box, function and derivative computation in general. As noted in Subsection [3.2.2], we are using

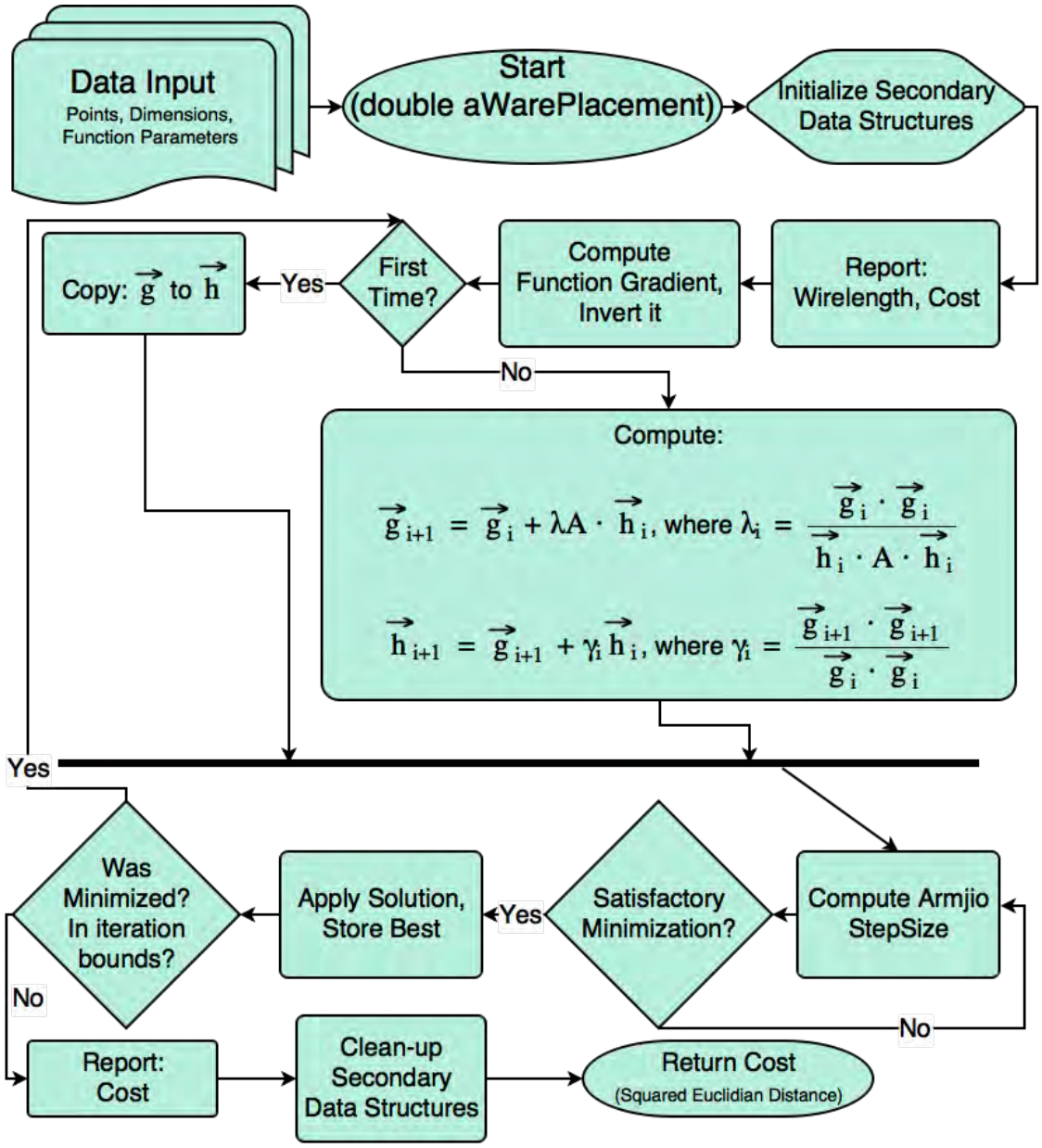

Figure 3.4: Minimization Logic

Sparse Matrices in order to handle the quadratic relationship of required RAM space and **IC** components to be minimized. In our novel solution, in order to keep the implementation as transparent as possible, and decoupled from external dependencies (to a logical extent), we refrained from using external libraries for our **Basic Linear Algebra Subprograms (BLAS)** Level 1 & 2 Operations. Additionally, our library providing the sparse matrices functionality (CXSparse), did not clearly list in its documentation the possibility to do a **BLAS** Level 2 Operation (sparse-matrix dense-vector multiplication).

Sparse-matrix dense-vector multiplication is an essential operation of our implementation since we operate on a sparse matrix and the coordinates of **IC**'s cells rest in a dense-vector. Moreover, since we continuously compute the function, the importance of its computation is increasingly paramount. Additionally, the provider of our "optimized" matrix (i.e. sparse matrix), should provide us said **BLAS** Operation. However, none of the aforementioned clauses happen - so everything is done manually, and it is not much optimized (mathematically or CPU architecture-wise).

Further intensifying the lack of optimal operations is the fact that instead of calculating the function's derivative analytically, we calculate it numerically, using the limit theorem. Instead of streamlining computation of all derivatives, and simplifying computations, we calculate the function twice, with really small differences in input parameters. Additionally, Armijo-Goldstein also continuously computes the input function, thus making function computation a critical performance factor in our proposed solution.

We could circumvent the majority of these problems (along with half of our thesis too), by moving our attempts to a mathematical software tool, for example, Matlab. Apart from an easier-to-use language, Matlab has years of experience handling mathematical and optimization problems, an asset that would heavily assist us. Our development efforts could be focused single-handedly only in the logic of the placement algorithm, disregarding all other parameters. Regrettably, this wasn't a plausible solution either, as there is comparable, if not more experience regarding all stages of **IC** design in our industrial **EDA** tool.

Chapter 4

Experimental Results

After describing our implementation, in detail, in this chapter, we will showcase our experimental results. We will provide all relevant minimization results and quality metrics - we will also discuss and give our personal view of the achieved results, objectively and with possible extensions.

4.1 Usage of Minimizer with Squared Euclidean cost function

Due to the issues described in the previous chapter, we were unable to benchmark any serious quantity of tests, or even attempt to run real benchmarking suites. We limited our testing only to small industrial **ICs**. Results can be seen on Table [4.1]. Note that negative "% QP Error" means that Solution provided is actually better than the QP **HPWL**. An example placement comparison between **aWarePlacement** and QP can be seen in Figures [4.1, 4.2].

	Bench #1	Bench #2	Bench #3	Bench #4
# Components	17	382	545	717
QP **HPWL**	47,664	1028,401	5524,569	9234,852
Armijo-Goldstein Parameters				
c	0,9	0,4	0,1	0,9
τ	0,5	0,8	0,8	0,5
Results				
Iterations	2159	15730	29028	13264
Total **HPWL**	46,016364	1027,910306	5524,726508	9232,987364
Quadratic Wirelength Cost	115,57196	5672,332454	64747,01338	130297,0628
CPU Time (s)	0,08	76,71	2250,63	153,9
QP Error	-3,4568%	-0,0477%	+0,0029%	-0,0202%

Table 4.1: Experimental results on benchmarks

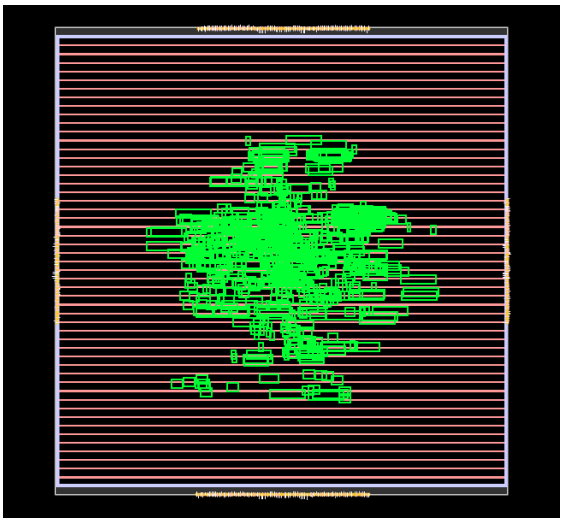

Figure 4.1: Example placement of **aWarePlacement** (*Benchmark #4*) ($c = 0.9, \tau = 0.5$)

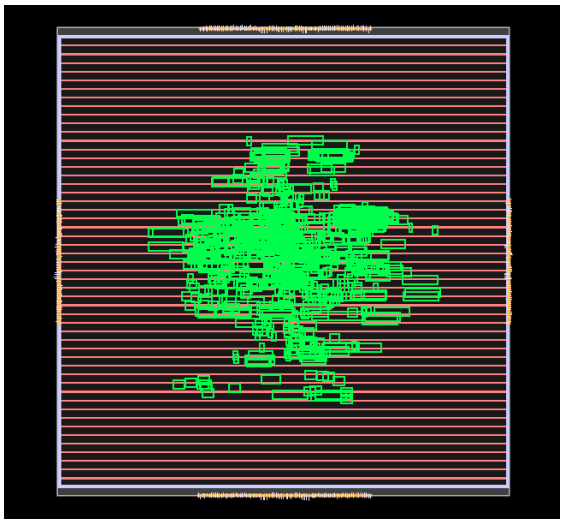

Figure 4.2: Example placement of **Quadratic Placement** (*Benchmark #4*)

The selection of our results was done using the minimum % QP Error. Although, using different metrics, or a weighted version of them which would be multiple times more profitable. Another option would be to attempt to unify the Armijo-Goldstein parameters, which has a vague pattern depending on selection and sorting parameters. However, it is being discussed in the next section.

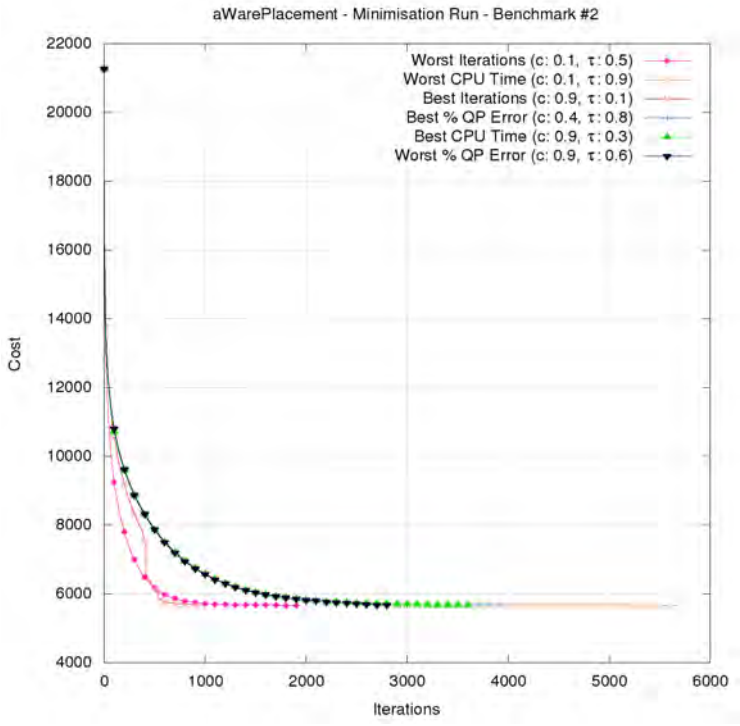

Figure 4.3: Example placement of Benchmark #2

It is fascinating to note that, this approach gives us a variety of input parameters for the Armijo-Goldstein algorithm (more on that in the next section), and also variety in the executed minimization iterations (and as such, time as well). Execution time is indeed not something to be proud of, but we have already addressed such concerns in the previous chapter. In Figure [4.3] we see a graph of execution of Benchmark #2 for various selected parameters: Best and Worst in Iterations, % QP Error and Execution Time. As we can see, apart from the Worst % QP Error, all other executions unravel about the same way. Although those cases are too close to call by simply watching a stationary, non-interactive image.

It is also possible that, if we accept less accuracy from the expected solution, then it is easier for us to save precious execution time. If we take a look at Figure [4.4], we can clearly see that the more tolerant on less accuracy we are, the returns on execution time are much greater. It should be noted that in a real application, we will not be able to have the solution to compare a result, so the loss of accuracy and execution time benefit are only for reference. Also, this change was done by solely tinkering with the Armijo-Goldstein c, τ parameters - not some innate error

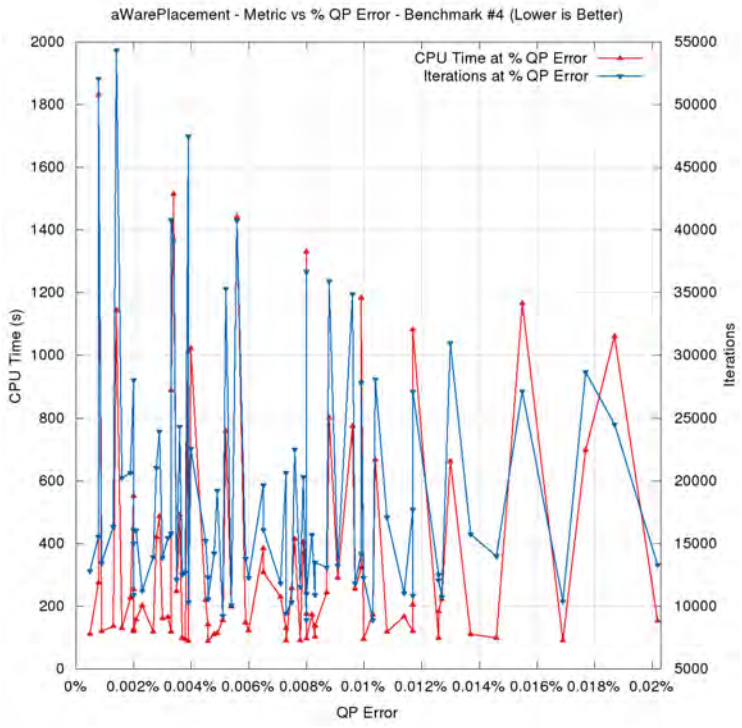

Figure 4.4: Metric vs % QP Error - Benchmark #4

tolerance parameter. We can clearly see that Armijo-Goldstein is a major factor in our algorithm that can save or crush any execution, depending on what your definition of "salvation" might be at the time. And that is exactly our field of focus in the next section.

4.2 Armijo-Goldstein Line Search

As Table 4.2 shows, our thoughts back in 2.7 actually hold true for the simple cases - a combination of a high τ and a low-to-medium c value indeed give us the best results. For clarity reasons, we state that the results are sorted in ascending % QP Error. Note that, while all presented set of values have reached their optimal solution (to the extent that we can objectively define what "optimal" means), this is not at all enforced. Some combinations gave good estimates, but they could have stopped preemptively, without a possibility to continue. *We duly note* that this hasn't happened in any of the results presented here (nor in their full matrix from where this excerpt was taken), but it *has* emerged on some conducted experiments.

Looking at the upper end of Table [4.2], we can see that this is also verified for larger and more complicated designs. However, this time, the optimized parameter in question rather than being the accuracy of the given results, now is the minimization of execution time. Specifically, the differences in the best solution and the worst is between 1 - 3 units, regardless of the actual returned result. Whether the results are in order of magnitude 1 or 4, the results are still within that area - in-correlatable with any parameter of the design (number of components, wirelength etc).

It might be that the results are close in returned cost, that is not the case with required iterations: the slowest with the fastest round are different by a factor of x4 to x8! That comes though, without an actual gain for solution quality. It is, again, in-correlatable the iteration cost of the various minimization parameters, and the returned quality of the results.

Although that the specifics of this pattern may not replicate in bigger test cases, we believe that it is a pattern that somehow will exist. Someone could devise a weighting metric that would take into account loss of accuracy from a predicted cost function, and CPU time / iteration count, and come up with a sorting method that would allow him / her to decipher the best Armijo-Goldstein parameters. Another course of action would be to blacklist parameters and then apply any logic to the results, should they are not straightforward to the eye.

Coming back to the point were we talked about emerging patterns in Armijo-Goldstein parameters, we have also plotted this matrix ascending in resulting function cost (and / or **HPWL**) and then ascending in iterations count. The matrix does look alike - specifically, values of c 0.1, τ [0.7 - 0.9] and c [0.7 - 0.9], τ [0.1 - 0.3] emerge as the most common to be higher up on our list. If someone were to be tolerant to a high-error sensitivity, no questions asked, then these are the values / patterns that emerge to the naked eye. As an adjacent pattern, we also see that in the pool of the top 10% of iterations, there is bound to be at least one of the top 10% cost metric (**HPWL**) results. Although not directly linked to the c, τ parameters, deciphering the reasoning behind low-iteration parameters, can subsequently lead to one of the best solutions

However, this is doesn't appear to be accurate enough or sufficiently answer all the questions asked. Given that the optimized function is indeed a convex function, *even if* Armijo-Goldstein performs differently per cost function, it doesn't explain why it performs differently with the same cost function. Of course, one explaination is that each **IC** has a different **Laplacian Matrix** and as such, shapes a wildly different function plane, and as such requires different descent parameters. However, I must note that this is just an impression, not a scientifically verified fact.

In finalizing this part of our research, it is worth noting that, results for any else than the aforementioned outliers, are about the same between the same group of results. The output is comparable, final cost and iterations. We couldn't identify a sophisticated pattern emerging from the raw data that we produced. We leave that as a point of a future study to discover the correlation between parameters and output data. It would also be interesting to study, given the parameter's correlation to the results, how a mix of those parameters would alter the actual algorithm in results and in time consumption.

4.2.1 Notes regarding Armijo-Goldstein Monte Carlo

To our knowledge, the provided results give little-to-no understanding in order to assist us in systematically leveraging the Armijo-Goldstein algorithm presented. There are combinations that can give best results, but our limited testing set (limited by the execution time caveats described before, not by lack of testing cases), prohibits us from doing any meaningful statistical analysis - or even machine learning analysis on the data. The only meaningful results we could see is the aforementioned. However, if it was possible to dynamically alter c, τ, we could see different or more concluding results.

4.3 Final Notes Regarding the Implementation

It was a challenging task to undergo, that had a lot of caveats, and needed bold moves to arrive at today's result. The most challenging topic of this thesis for me was, by far, the mathematical models, theories, and data structures. It was required

x^2 + y^2		Beale		Bench #1 (17)			Bench #2 (382)			Bench #3 (545)			Bench #4 (717)		
c	τ	c	τ	c	τ	% QP Error	c	τ	% QP Error	c	τ	% QP	c	τ	% QP Error
0,6	0,1	0,7	0,1	0,9	0,5	-3,45677%	0,4	0,8	-0,04771%	0,1	0,8	0,00285%	0,9	0,5	-0,02019%
0,9	0,1	0,8	0,1	0,9	0,9	-2,92956%	0,8	0,5	-0,04186%	0,1	0,9	0,00298%	0,8	0,9	-0,01870%
0,8	0,1	0,8	0,2	0,9	0,4	-2,86846%	0,7	0,7	-0,04009%	0,1	0,6	0,00299%	0,7	0,8	-0,01773%
0,3	0,1	0,7	0,2	0,9	0,2	-2,83465%	0,6	0,3	-0,03574%	0,1	0,5	0,00300%	0,9	0,2	-0,01689%
0,4	0,1	0,4	0,1	0,8	0,4	-2,31704%	0,5	0,1	-0,03423%	0,1	0,2	0,00302%	0,6	0,9	-0,01548%
0,5	0,1	0,6	0,1	0,9	0,6	-2,26148%	0,5	0,4	-0,03085%	0,1	0,7	0,00304%	0,9	0,3	-0,01464%
0,3	0,2	0,5	0,1	0,9	0,7	-2,21055%	0,4	0,9	-0,03047%	0,1	0,3	0,00304%	0,7	0,2	-0,01368%
0,7	0,1	0,6	0,2	0,9	0,8	-2,16833%	0,5	0,2	-0,02968%	0,1	0,4	0,00305%	0,8	0,8	-0,01304%
0,7	0,2	0,9	0,2	0,7	0,5	-2,10207%	0,5	0,8	-0,02945%	0,1	0,1	0,00306%	0,8	0,1	-0,01265%
0,9	0,2	0,5	0,2	0,6	0,5	-2,08489%	0,9	0,1	-0,02891%	0,2	0,6	0,00563%	0,6	0,6	-0,01262%
0,9	0,4	0,9	0,1	0,5	0,7	-2,01954%	0,8	0,1	-0,02863%	0,2	0,4	0,00575%	0,5	0,3	-0,01261%
0,8	0,2	0,1	0,2	0,8	0,9	-1,95616%	0,9	0,3	-0,02812%	0,2	0,8	0,00586%	0,7	0,9	-0,01175%

Table 4.2: Monte Carlo: Armijo-Goldstein c, τ parameters (clipped)

to delve deep and familiarize with them in a great deal in order to utilize them in a consortium to achieve a solution in reasonable execution time.

4.3.1 Mathematical Understanding

There were a lot of misunderstandings and wrong assumptions that were made during the development process. The replication of the mathematical aspect in the code was extremely hard, and, to my understanding, it is not yet quite done. According to literature, we would need $O(N)$ calculations of the function in the worst case, but that is by far different from what our implementation does. I can only assume that the issue lies with the implementation not being optimal, or also it could be that our problem is much too complex to be solved as such with this implementation. Finally, one major factor could also be that I personally lack complete understanding of the involved methods

4.3.2 Mathematical Operations

Another issue with my implementation, regarding computation time, is that function computation is expensive with regard to what computations are made. Apart from the necessary iteration of the I/O pins, all operations are **BLAS** Operations - so that would be a perfect candidate for optimization, which is also already studied and optimized even per-architecture. However, our library providing the sparse matrices (CXSparse) does not have enough documentation - and we haven't found a method to optimize sparse-matrix dense-vector multiplication, which is a major component of the function, for a lot of iterations (more than those reported) in the results. As such, we have to manually iterate the whole sparse matrix and compute the multiplication as normal. We have an optimization in-place, making the multiplication only based on the Coordinate List, instead of iterating the sparse matrix as normal. Unfortunately, this is not enough to help us support high cell count and benchmarking suites.

In addition to not having optimal function computation, we have to face the fact that instead of an optimal derivative computation, i.e. analytically, we

calculate the numerical derivative. This also introduces a slow down in execution, as every dimension needs 2 function computation invocations with almost similar parameters. Additionally, it introduces an error by an order of h^2, according to literature. Although the error can be minimized by selecting h as the quantum of each dimension - which is also verified by Matlab's gradient computations - the hit on execution time is really significant. It is a good thing that we do not call the gradient computation too much - which is the only alternative "optimization" other than optimizing the function itself.

4.3.3 Other Free Parameters of the Implementation

Inner Iteration Limit

In our flawed implementation, after testing, we verified that the execution time can vary greatly depending on the limit of inner iterations we allow our algorithm to execute. Allowing a smaller number of iterations usually results in smaller execution time. Tests show that after a number of iterations, the inner algorithm, rather than determining approximate positions for all the components, it fine-tunes the current solution. This is a good quality to have in the later stages of the algorithm - although it could be argued that it is not completely necessary for 2 reasons:

- Fine-tuning at the beginning of the algorithm, at the point where algorithm is most volatile, is of no use, since the solution will be quickly overwritten
- Fine-tuning the solution with changes that do not accumulate to more than $\frac{1}{2}^{th}$ of the dimension's quantum will not affect at all any stage of the **EDA** flow; **Legalization**, **Placement** or otherwise

While finely-tuned solutions are a good quality to have, we can argue that we are way too early in the Physical Design stage for any accuracy to matter - especially when accuracy is quantized!

Of course, there is a limit on how small that limit should be because otherwise, we are not allowing the algorithm to execute its minimization process correctly. However, there is no apparent correlation between any of the problem's parameters:

We discovered through testing that a good number for a small limit lies within the $[150, 250]$ span.

Outer Iteration Limit

In a similar manner to the "Inner Iteration Limit", "Outer Iteration Limit" is behaving in a recognizable pattern. In this case, luckily, early iterations are mostly volatile and not so much fine. However, the algorithm, in the end, can (and will) spend extra time fine-tuning the solution. This is getting delayed further by continuous outer invocations, which are more expensive than inner ones. That is further augmented by the fact that inner iterations tend to be a lot fewer on the final stages of the solution, thus creating even more outer invocations. We haven't tested this feature at all, since our developing attempts in this thesis appear to be closely related to "proof of concept", rather than "optimization / breakthrough".

Chapter 5

Conclusions and Future Work

In this work, we proposed an evolution of the **Analytical Global Placement** part of the well-known NTUPlace3 [Che+08] algorithm and its aim was to propose a fully autonomous minimizing implementation, with an arbitrary cost function as input. Here is the summary of my whole thesis.

Global Placement is the first one of the three tasks of standard cell **Placement**. **Global Placement**, which aims to generate a "rough" placement solution that completely disregards all **IC** rules for **Placement** - most importantly cell overlap and cell alignment to **Placement** rows and columns. This was our focus. The second placement task, **Legalization**, is the process that takes into account all previously skipped **IC** regulations (overlap and alignment). The last task, **Detailed Placement**, further improves the **Legalization Placement** solution.

Global Placement's target is to initialize the positions of the cells, as optimally as possible, attempting to adhere to as many rules as possible, and, preferably not monopolize execution time. There is an entire flow waiting to execute after **Global Placement**, which are not a subset of this algorithm.

There are multiple ways to do this: using **Combinatorial** or **Analytical** optimization. The latter is used in this case, and it means we are using a mathematical cost function to achieve superb placement, taking into account as much restrictions as possible. What this also means is that, after formulating said function, then accelerating its minimization is problem-agnostic - so we can freely

apply any and all mathematical tools at our disposal. This is where Armijo-Goldstein Line Search comes into play to speed up the minimization.

To sum up, **aWarePlacement** was successful in optimizing our set goals. The strength of the solution is mainly on the part that it is really modular with respect to input arguments. It is compromised of mathematical methods that ensure deterministic behavior, rapid convergence, isolation between algorithm's components. Moreover, cluster support is included in order to sustain designs of arbitrary size. Its greatest weakness is, however the execution time for the reasons we have already discussed: we are doing multiple unoptimized operations at the code level and we are missing the mathematical background to leverage all the computationally-faster methods than the ones we are using.

aWarePlacement can be easily extended and optimized in the future, using the following guidelines:

- Accompany Armijo-Goldstein with Golden-Section and Quadratic Euclidean Movement Bound and compare them
- Analytical Function Derivative
- Clean, documented, and tested sparse matrix API
- Discover the logic of Armijo-Goldstein's optimal parameters (c, τ)
- Accompanied in Timing Driven **Placement**
- Since we plan to use the minimizer with non-convex functions, it would be appropriate to extend the minimizer with hill-climbing methodologies.
- Expanded to be used in 3D Placement
- Implement Clusters Support
- Combination with a **Legalizer**, like *Abacus2*, to achieve iteratively better quality results

Bibliography

[SS85] Carl Sechen and Alberto Sangiovanni-Vincentelli. "The TimberWolf placement and routing package". In: *IEEE Journal of Solid-State Circuits* 20.2 (1985), pp. 510–522.

[Kle+91] Jürgen M Kleinhans et al. "GORDIAN: VLSI placement by quadratic programming and slicing optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.3 (1991), pp. 356–365.

[YCS03] Xiaojian Yang, Bo-Kyung Choi, and M. Sarrafzadeh. "Routability-driven white space allocation for fixed-die standard-cell placement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.4 (Apr. 2003), pp. 410–419. ISSN: 0278-0070. DOI: 10.1109/TCAD.2003.809660.

[KW05] Andrew B Kahng and Qinke Wang. "Implementation and extensibility of an analytic placer". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.5 (2005), pp. 734–747.

[Luc06] Grant Martin Luciano Lavagno Louis Scheffer. *EDA for IC implementation, circuit design, and process technology*. 1st ed. Electronic design automation for integrated circuits handbook. CRC Taylor & Francis, 2006. ISBN: 0849379245, 9780849379246, 0849330963.

[Pre+07] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688.

[Cha08] Sachin S. Sapatnekar Charles J. Alpert Dinesh P. Mehta. *Handbook of Algorithms for Physical Design Automation*. 1st ed. 2008. ISBN: 0849372429,9780849372421,9781420013481.

[Che+08] Tung-Chieh Chen et al. "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1228–1240. URL: http://ieeexplore.ieee.org/document/4544855/.

[Pbr08] Pbroks13. *File:X-intercepts.svg — Wikimedia Commons*. SVG redraw of [[:Image:X-intercepts.PNG]] | Source= http://en.wikipedia.org/wiki/Image:X-intercepts.PNG; accessed 26-Apr-2017. 2008. URL: https://commons.wikimedia.org/wiki/File:X-intercepts.svg.

[Kah+11] Andrew B Kahng et al. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[Lin12] Linear77. *File:PhysicalDesign.png — Wikimedia Commons*. [English: VLSI circuit design flow with a focus on physical design, Deutsch: Schritte des Layoutentwurfs innerhalb des Chipentwurfs. Online; accessed 06-Jan-2017]. 2012. URL: https://commons.wikimedia.org/wiki/File:PhysicalDesign.png.

[JY13] Momin Jamil and Xin-She Yang. "A literature survey of benchmark functions for global optimisation problems". In: *International Journal of Mathematical Modelling and Numerical Optimisation* 4.2 (2013), pp. 150–194. URL: https://arxiv.org/pdf/1308.4008.pdf.

[Ade14] Kenneth C. Smith Adel S. Sedra. *Microelectronic Circuits*. 7th ed. The Oxford Series in Electrical and Computer Engineering. Oxford University Press, 2014. ISBN: 0199339139,9780199339136.

[VM16] Xanthos Vlachos and Giaourtas Mixalis. "Implementation and Analysis of Placement Algorithms based on methods of Mechanics (Force Directed) for Microelectronic Circuits". In: (Oct. 2016).

[Mer17] Merriam-Webster, ed. *Define Electrical for English Language Learners*. Jan. 2, 2017. URL: https://www.merriam-webster.com/dictionary/electronic.

[Wik17a] Wikipedia. *Backtracking line search — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-January-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Backtracking_line_search&oldid=759866180.

[Wik17b] Wikipedia. *Basic Linear Algebra Subprograms — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-June-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Basic_Linear_Algebra_Subprograms&oldid=777442365.

[Wik17c] Wikipedia. *Combinatorial optimization — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-April-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Combinatorial_optimization&oldid=770375802.

[Wik17d] Wikipedia. *Laplacian matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Laplacian_matrix&oldid=777613302.

[Wik17e] Wikipedia. *Mathematical optimization — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-April-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Mathematical_optimization&oldid=771950307.

[Wik17f] Wikipedia. *Sparse matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=777209696.

[Wik17g] Wiktionary. *Manhattan distance — Wiktionary, The Free Dictionary*. [Online; accessed 7-April-2017]. 2017. URL: https://en.wiktionary.org/w/index.php?title=Manhattan_distance&oldid=42263941.

List of Figures

List of Tables

Acronyms

BLAS Basic Linear Algebra Subprograms. (Pages 33, 42, *Glossary:* Basic Linear Algebra Subprograms)

CG Conjugate Gradient. (Pages 9, 14, 17, 18, *Glossary:* Conjugate Gradient)

EDA Electronic Design Automation. (Pages 1, 2, 19, 21, 23, 27, 33, 43, *Glossary:* Electronic Design Automation)

HPWL Half-Perimeter WireLength. (Pages 5, 10, 11, 14, 20, 21, 35, 36, 39, *Glossary:* Half-Perimeter WireLength)

IC Integrated Circuit. (Pages 1–4, 13, 20, 22, 28, 29, 33, 35, 40, 45, 59, *Glossary:* Integrated Circuit)

VLSI Very-Large-Scale Integration. (Pages 1, *Glossary:* Very-Large-Scale Integration)

Glossary

Analytical (**optimization**) see: **Mathematical** (Pages 5–7, 9, 10, 12, 15, 18, 29, 30, 45)

Basic Linear Algebra Subprograms (**BLAS**) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. They are the de facto standard low-level routines for linear algebra libraries; the routines have bindings for both C and Fortran. Although the **BLAS** specification is general, **BLAS** implementations are often optimized for speed on a particular machine, so using them can bring substantial performance benefits. **BLAS** implementations will take advantage of special floating point hardware such as vector registers or SIMD instructions.

[...]

Functionality

BLAS functionality is categorized into three sets of routines called "levels", which correspond to both the chronological order of definition and publication, as well as the degree of the polynomial in the complexities of algorithms; Level 1 BLAS operations typically take linear time $O(n)$, Level 2 operations quadratic time and Level 3 operations cubic time. Modern BLAS implementations typically provide all three levels.

Level 1

This level consists of all the routines described in the original presentation of BLAS (1979), which defined only vector operations on strided arrays: dot products, vector norms, a generalized vector addition of the form $\boldsymbol{y} \leftarrow \alpha\boldsymbol{x} + \boldsymbol{y}$ (called "axpy") and several other operations.

Level 2

This level contains matrix-vector operations including, among other things, a generalized matrix-vector multiplication (`gemv`): $\boldsymbol{y} \leftarrow \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$ as well as a solver for x in the linear equation $\boldsymbol{T}\boldsymbol{x} = \boldsymbol{y}$ with T being triangular. Design of the Level 2 BLAS started in 1984, with results published in 1988. The Level 2 subroutines are especially intended to improve performance of programs using BLAS on vector processors, where Level 1 BLAS are suboptimal "because they hide the matrix-vector nature of the operations from the compiler".

Level 3

This level, formally published in 1990, contains matrix-matrix operations, including a "general matrix multiplication" (`gemm`), of the form $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$ where A and B can optionally be transposed or hermitian-conjugated inside the routine and all three matrices may be strided. The ordinary matrix multiplication AB can be performed by setting α to one and C to an all-zeros matrix of the appropriate size.

Also included in Level 3 are routines for solving $\boldsymbol{B} \leftarrow \alpha\boldsymbol{T}^{-1}\boldsymbol{B}$ where T is a triangular matrix, among other functionality. [Wik17b] (Page 33)

Combinatorial (**optimization**) is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not feasible. It operates on the domain of those optimization problems, in which the set of feasible solutions is discrete or can be reduced to discrete, and in which the goal is to find the best solution. Some common problems involving combinatorial optimization are the Traveling Salesman Problem ("TSP") and the Minimum Spanning Tree ("MST") problem. [Wik17c] (Pages 5, 6, 45)

Conjugate Gradient (**method**) is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. The conjugate gradient method is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition (Page 9)

Critical path is defined as the path between an input and an output with the maximum delay. (Page 4)

Electronic Design Automation (**EDA**), also referred to as **Electronic Computer-Aided Design** (**ECAD**), is a category of software tools for designing electronic systems such as integrated circuits and printed circuit boards. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips. Since a modern semiconductor chip can have billions of components, EDA tools are essential for their design (Page 1)

Half-Perimeter WireLength The half-perimeter wirelength model is commonly used because it is reasonably accurate and efficiently calculated. The *bounding box* of a net with p pins is the smallest rectangle that encloses the pin locations. The wirelength is estimated as half the perimeter of the bounding box. For two- and three-pin nets (70-80% of all nets in most modern designs), this is exactly the same as the rectilinear Steiner minimum tree (RSMT) cost (discussed later in [Kah+11]). When $p \geqslant 4$, HPWL underestimates the RSMT cost by an average factor that grows asymptotically as $\sqrt{p}$. [Kah+11, p. 97, s. 4.2] (Pages 5, 10)

Laplacian Matrix In the mathematical field of graph theory, the **Laplacian matrix** (sometimes called **admittance matrix**, **Kirchhoff matrix** or **discrete Laplacian**) is a matrix representation of a graph. Given a simple graph G with n vertices, its Laplacian matrix $L_{n\times n}$ is defined as:

$$L = D - A$$

where D is the degree matrix and A is the adjacency matrix of the graph. Since G is a simple graph, A only contains 1s or 0s and its diagonal elements are all 0s. In the case of directed graphs, either the in-degree or out-degree might be used, depending on the application. The elements of L are given by

$$L_{i,j} := \begin{cases} deg(v_i), & i = j \text{ where } deg(v_i) \text{ is the degree of vertex } i \\ -1, & i \neq j, \text{ and } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise} \end{cases}$$

[Wik17d] (Pages 21, 22, 29, 40)

Legalizer A **Placement legalizer** snaps cells to the sites of rows such that no cells overlap. This has to be done with minimum adverse impact on the quality of the placement. [Luc06] (Pages 5, 9, 10, 43, 45, 46, 59)

Manhattan Distance The distance between two points in a grid based on a strictly horizontal and/or vertical path (that is, along the grid lines), as opposed to the diagonal or "as the crow flies" distance. The Manhattan distance is the simple sum of the horizontal and vertical components, whereas the diagonal distance might be computed by applying the Pythagorean theorem. [Wik17g] (Pages 4, 20)

Mathematical (optimization) (alternatively named **mathematical programming** or simply **optimization** or **optimisation**), is the selecstion of a best element (with regard to some criterion) from some set of available alternatives.

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. [Wik17e] (Page 55)

Placement is an essential step in electronic design automation - the portion of the physical design flow that assigns exact locations for various circuit components within the chip's core area. An inferior placement assignment will not only affect the chip's performance but might also make it nonmanufacturable by producing excessive wirelength. Currently, placement is usually separated into **Global Placement**, **Legalizer** and ***Detailed Placement*** (Pages 2, 3, 5–10, 12, 13, 15, 18, 29, 43, 45, 46, 51, 58)

Switching activity is the measurement of changes of signal values. It has two parts: probability - the likelihood that a signal will have the logic value of '1' - and toggle density - the number of switches per unit time. (Page 4)

Very-Large-Scale Integration is the process of creating an **IC** by combining thousands of transistors into a single chip. The microprocessor is a VLSI device. Before the introduction of VLSI technology most **ICs** had a limited set of functions they could perform. An electronic circuit might consist of a CPU, ROM, RAM and other glue logic. **VLSI** lets **IC** designers add all of these into one chip. (Page 1)